Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

SOFTWARE TESTING

Principles and Practices

Naresh Chauhan

OXFORD
UNIVERSITY PRESS

© Oxford University Press 2010

ISBN: 978-0-1980618-47

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

To
my parents
who have made me capable
to struggle in this world

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Preface

There is no life without struggles and no software without bugs. Just as one needs to sort out the problems in one's life, it is equally important to check and weed out the bugs in software. Bugs cripple the software in a way problems in life unsettle one. In our life, both joys and sorrows are fleeting. But a person is best tested in times of crises. One who cultivates an optimistic outlook by displaying an equipoise taking prosperity as well as adversity in his stride and steadily ventures forth on a constructive course is called a sthir pragna. We should follow the same philosophy while testing software too. We need to develop an understanding that unless these bugs appear in our software and until we weed out all of them, our software will not be robust and of superior quality. So, a software test engineer should be an optimist who welcomes the struggles in life and similarly bugs in software, and takes them head-on.

Software engineering as a discipline emerged in the late 1960s to guide software development activities in producing quality software. Quality here is not a single-dimensional entity. It has several factors including rigorous software testing. In fact, testing is the critical element of quality and consumes almost half the total development effort. However, it is unfortunate that the quality and testing process does not get its due credit. In software engineering, testing is considered to be a single phase operation performed only after the development of code wherein bugs or errors are removed. However, this is not the case. Testing is not just an intuitive method to remove the bugs, rather it is a systematic process such as software development life cycle (SDLC). The testing process starts as soon as the first phase of SDLC starts. Therefore, even after learning many things about software engineering, there are still some questions and misconceptions regarding the testing process which need to be known, such as the following:

- When should testing begin?
- How much testing is practically possible?
- What are the various techniques to design a good test case (as our knowledge is only limited to black-box and white-box techniques)?

Moreover, the role of software testing as a systematic process to produce quality software is not recognized on a full scale. Many well-proven methods are largely unused in industries today. Companies rely only on the automated testing tools rather than a proper testing methodology. What they need to realize is that Computer-Aided Software Engineering (CASE) environments or tools are there only to assist in the development effort and not meant to serve as silver bullets! Similarly, there are many myths that both students and professionals believe in, which need to be exploded. The present scenario requires *software testing* to be acknowledged as a separate discipline from software engineering. Some universities have already started this course. Therefore, there is a need for a book that explains all these issues for the benefit of students who will learn software testing and become knowledgeable test engineers as also for the benefit of test engineers who are already working in the industries and want to hone their testing skills.

About the book

This book treats software testing as a separate discipline to teach the importance of testing process both in academia as well as in the industry. The book stresses on software testing as a systematic process and explains software testing life cycle similar to SDLC and gives insight into the practical importance of software testing. It also describes all the methods/techniques for test case design which is a prime issue in software testing. Moreover, the book advocates the notion of effective software testing in place of exhaustive testing (which is impossible).

The book has been written in a lucid manner and is packed with practical approach of designing the test cases targeting undergraduate and postgraduate students of computer science and engineering (B.Tech., M.Tech., MCA), and test engineers. It discusses all the software testing issues and gives insight into their practical importance. Each chapter starts with the learning objectives and ends with a summary containing a quick review of important concepts discussed in the chapter. Some chapters provide solved examples in between the theory to understand the method or technique practically at the same moment. End-chapter exercises and multiple-choice questions are provided to assist instructors in classroom teaching and students in preparing better for their exams.

The key feature of the book is a fully devoted case study on Income Tax Calculator which shows how to perform verification and validation at various phases of SDLC. The case study includes ready-to-use software and designing of test cases using the techniques described in the book. This material will help both students and testers understand the test design techniques and use them practically.

Apart from the above-mentioned features, the book follows the following methodology in defining key concepts in software testing:

- Emphasis on software testing as a systematic process
- Effective testing concepts rather than exhaustive complete testing
- A testing strategy with a complete roadmap has been developed that shows which software testing technique with how much risk assessment should be adopted at which phase of SDLC
- Testing models
- Verification and validation as the major components of software testing process. These have been discussed widely expanding in separate chapters.
- Software testing life cycle along with bug classification and bug life cycle
- Complete categorization of software testing techniques such as static testing and dynamic testing expanding in different chapters
- Testing techniques with solved examples to illustrate how to design test cases using these techniques
- Extensive coverage of regression testing, software testing metrics, and test management
- Efficient test suite management to prioritize test cases suitable for a project
- The appropriate use of testing tools
- Software quality management and test maturity model (TMM)
- Testing techniques for two specialized environments: object-oriented software and Web-based software

About the CD

The CD accompanying the book contains the following:

- *Executable files* for the examples given in Chapter 5 so that a user can directly implement white-box testing on the codes without any extra effort.
- *Checklists* for verification of parameters, such as general software design document (SDD), generic code, high level design (HLD), low level design (LLD), and software requirement specification (SRS) document.
- *A program on Income Tax Calculator* along with its description in the form of a case study that illustrates all the steps of the software testing process.

Content and Coverage

The book has been divided into seven different parts. Each part further consists of various chapters.

Part I (Testing Methodology) introduces concepts such as effective software testing, testing terminology, testing as a process, and development of testing methodology.

Chapter 1 introduces the concept of effective testing versus complete testing, explains the psychology for performing effective testing, and establishes that software testing is a complete process.

Chapter 2 discusses the commonly used testing terminology such as error, bug, and failure, explains life cycle of a bug with its various states, phases of software testing life cycle and V testing model, and development of a testing methodology.

Chapter 3 explains how verification and validation, a part of testing strategy, are performed at various phases of SDLC.

Part II (Testing Techniques) deals with various test case design techniques based on static testing and dynamic testing and verification and validation concepts.

Chapter 4 covers test case design techniques using black-box testing including boundary value analysis, equivalence class partitioning method, state table based testing, decision table based testing, and cause-effect graphing technique.

Chapter 5 discusses test case design techniques using white-box testing, including basis path testing, loop testing, data flow testing, and mutation testing.

Chapter 6 deals with the techniques, namely inspection, walkthrough, and reviews, largely used for verification of various intermediate work products resulting at different stages of SDLC.

Chapter 7 discusses various techniques used in validation testing such as unit testing, integration testing, function testing, system testing, and acceptance testing.

Chapter 8 describes regression testing that is used to check the effect of modifications on other parts of software.

Part III (Managing the Testing Process) discusses how to manage the testing process, various persons involved in test organization hierarchy, testing metrics to monitor and control the testing process, and how to reduce the number of test cases.

Chapter 9 covers the concept of introduction of management for test process for its effectiveness. Various persons involved in test management hierarchy are discussed. The test planning

for various verification and validation activities are also discussed along with the test result specifications.

Chapter 10 provides an introductory material to understand that measurement is a necessary part of software engineering, known as software metrics.

Chapter 11 explains how software metrics assist in monitoring and controlling different testing activities.

Chapter 12 explains the fact that test cases, specially designed for system testing and regression testing, become unmanageable in a way that we cannot test all of them. The problem is how to select or reduce the test cases out of a big test suite. This chapter discusses many such techniques to resolve the problem.

Part IV (Quality Management) covers software quality issues with some standards along with testing process maturity models.

Chapter 13 discusses various terminologies, issues, and standards related to software quality management to produce high quality software.

Chapter 14 discusses various test process maturity models, namely test improvement model (TIM), test organization model (TOM), test process improvement (TPI), and test maturity model (TMM).

Part V (Test Automation) discusses the need of testing and provides an introduction to testing tools.

Chapter 15 explains the need for automation, categories of testing tools, and how to select a testing tool.

Part VI (Testing for Specialized Environment) introduces the testing environment and the issues related to two specialized environments, namely object-oriented software and Web-based software.

Chapters 16 and 17 discuss the issues, challenges, and techniques related to object-oriented and Web-based software, respectively.

Part VII (Tracking the Bug) explains the process and techniques of debugging.

Chapter 18 covers the debugging process and discusses various methods to debug a software product.

The book concludes with a case study of Income Tax Calculator illustrating the testing of software using verification and validation techniques. In addition, the book contains useful appendices that provide various ready-to-use checklists which can be used at the time of verification of an item at various stages of SDLC.

Do send your valuable suggestions, comments, and constructive criticism for further improvement of the book.

Naresh Chauhan

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Contents

Preface *v*

PART 1: Testing Methodology

1. Introduction to Software Testing 3

1.1 Introduction *3*
1.2 Evolution of Software Testing *5*
1.3 Software Testing–Myths and Facts *8*
1.4 Goals of Software Testing *10*
1.5 Psychology for Software Testing *13*
1.6 Software Testing Definitions *14*
1.7 Model for Software Testing *15*
1.8 Effective Software Testing vs. Exhaustive Software Testing *16*
1.9 Effective Testing is Hard *21*
1.10 Software Testing as a Process *22*
1.11 Schools of Software Testing *23*
1.12 Software Failure Case Studies *25*

2. Software Testing Terminology and Methdology 32

2.1 Software Testing Terminology *33*
2.2 Software Testing Life Cycle (STLC) *46*
2.3 Software Testing Methodology *51*

3. Verification and Validation 65

3.1 Verification and Validation (V&V) Activities *66*
3.2 Verification *69*
3.3 Verification of Requirements *70*
3.4 Verification of High-level Design *74*
3.5 Verification of Low-level Design *76*
3.6 How to Verify Code? *77*
3.7 Validation *79*

PART 2: Testing Techniques

4. Dynamic Testing: Black-Box Testing Techniques 89

4.1 Boundary Value Analysis (BVA) *90*
4.2 Equivalence Class Testing *107*
4.3 State Table-Based Testing *114*
4.4 Decision Table-Based Testing *119*
4.5 Cause-Effect Graphing Based Testing *125*
4.6 Error Guessing *129*

5. Dynamic Testing: White-Box Testing Techniques 135

5.1 Need of White-Box Testing *135*
5.2 Logic Coverage Criteria *136*
5.3 Basis Path Testing *138*
5.4 Graph Matrices *156*
5.5 Loop Testing *161*
5.6 Data Flow Testing *164*
5.7 Mutation Testing *174*

6. Static Testing 188

6.1 Inspections *190*
6.2 Structured Walkthroughs *205*
6.3 Technical Reviews *206*

7. Validation Activites 212

7.1 Unit Validation Testing *213*
7.2 Integration Testing *218*
7.3 Function Testing *231*
7.4 System Testing *233*
7.5 Acceptance Testing *244*

8. Regression Testing 255

8.1 Progressive vs. Regressive Testing *255*
8.2 Regression Testing Produces Quality Software *256*
8.3 Regression Testability *257*
8.4 Objectives of Regression Testing *258*
8.5 When is Regression Testing Done? *258*
8.6 Regression Testing Types *259*
8.7 Defining Regression Test Problem *259*
8.8 Regression Testing Techniques *260*

PART 3: Managing the Testing Process

9. Test Management 273

9.1 Test Organization *274*
9.2 Structure of Testing Group *275*
9.3 Test Planning *276*
9.4 Detailed Test Design and Test Specifications *292*

10. Software Metrics 304

10.1 Need of Software Measurement *305*
10.2 Definition of Software Metrics *306*
10.3 Classification of Software Metrics *306*
10.4 Entities to be Measured *307*
10.5 Size Metrics *308*

11. Testing Metrics for Monitoring and Controlling the Testing Process 317

11.1 Measurement Objectives for Testing *318*
11.2 Attributes and Corresponding Metrics in Software Testing *319*
11.3 Attributes *320*
11.4 Estimation Models for Estimating Testing Efforts *327*
11.5 Architectural Design Metric Used for Testing *331*
11.6 Information Flow Metrics Used for Testing *332*
11.7 Cyclomatic Complexity Measures for Testing *333*
11.8 Function Point Metrics for Testing *333*
11.9 Test Point Analysis (TPA) *335*
11.10 Some Testing Metrics *341*

12. Efficient Test Suit Management 352

12.1 Why Does a Test Suite Grow? *352*
12.2 Minimizing the Test Suite and its Benefits *353*
12.3 Defining Test Suite Minimization Problem *354*
12.4 Test Suite Prioritization *354*
12.5 Types of Test Case Prioritization *355*
12.6 Prioritization Techniques *356*
12.7 Measuring the Effectiveness of a Prioritized Test Suite *365*

PART 4: Quality Management

13. Software Quality Management 373

13.1 Software Quality *374*
13.2 Broadening the Concept of Quality *374*
13.3 Quality Cost *375*
13.4 Benefits of Investment on Quality *376*
13.5 Quality Control and Quality Assurance *377*
13.6 Quality Management (QM) *378*
13.7 QM and Project Management *379*
13.8 Quality Factors *379*
13.9 Methods of Quality Management *380*
13.10 Software Quality Metrics *387*
13.11 SQA Models *390*

14. Testing Process Maturity Models 404

14.1 Need for Test Process Maturity *405*
14.2 Measurement and Improvement of a Test Process *406*
14.3 Test Process Maturity Models *406*

PART 5: Test Automation

15. Automation and Testing Tools 429

15.1 Need for Automation *430*
15.2 Categorization of Testing Tools *431*
15.3 Selection of Testing Tools *434*

15.4 Costs Incurred in Testing Tools *435*
15.5 Guidelines for Automated Testing *436*
15.6 Overview of Some Commercial Testing Tools *437*

PART 6: Testing for Specialized Environment

16. Testing Object-Oriented Software 445
16.1 OOT Basics *446*
16.2 Object-oriented Testing *450*

17. Testing Web-based Systems 474
17.1 Web-based System *474*
17.2 Web Technology Evolution *475*
17.3 Traditional Software and Web-based Software *476*
17.4 Challenges in Testing for Web-based Software *477*
17.5 Quality Aspects *478*
17.6 Web Engineering (Webe) *480*
17.7 Testing of Web-based Systems *484*

PART 7: Tracking the Bug

18. Debugging 503
18.1 Debugging: an Art or Technique? *503*
18.2 Debugging Process *504*
18.3 Debugging Is Difficult *505*
18.4 Debugging Techniques *506*
18.5 Correcting the Bugs *509*
18.6 Debuggers *510*

Income Tax Calculator: A Case Study

Step 1 *Introduction to Case Study* *513*
Step 2 *Income Tax Calculator SRS ver 1.0* *515*
Step 3 *Verification on Income Tax Calculator SRS ver 1.0* *517*
Step 4 *Income Tax Calculator SRS ver 2.0* *520*
Step 5 *Verification on Income Tax Calculator SRS ver 2.0* *525*
Step 6 *Income Tax Calculator SRS ver 3.0* *531*
Step 7 *Black-Box Testing on Units/Modules of Income Tax Calculator SRS ver 3.0* *538*
Step 8 *White-Box Testing on Units/Modules of Income Tax Calculator* *552*

Appendices

Appendix A *Answers to Multiple Choice Questions* *587*
Appendix B *Software Requirement Specification (SRS) Verification Checklist* *589*
Appendix C *High Level Design (HLD) Verification Checklist* *592*
Appendix D *Low Level Design (LLD) Verification Checklist* *594*
Appendix E *General Software Design Document (SDD) Verification Checklist* *595*
Appendix F *Generic Code Verification Checklist* *596*

References *600*
Index *606*

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Part 1 Testing Methodology

CHAPTERS

Chapter 1:
Introduction to Software Testing

Chapter 2:
Software Testing Terminology and Methodology

Chapter 3:
Verification and Validation

Software testing has always been considered a single phase performed after coding. But time has proved that our failures in software projects are mainly due to the fact that we have not realized the role of software testing as a process. Thus, its role is not limited to only a single phase in the software development life cycle (SDLC), but it starts as soon as the requirements in a project have been gathered.

Complete software testing has also been perceived for a long time. Again, it has been proved that exhaustive testing is not possible and we should shift our attention to effective testing. Thus, effective and early testing concepts build our testing methodology. Testing methodology shows the path for successful testing. This is the reason that parallel to SDLC, software testing life cycle (STLC) has also been established now.

The testing methodology is related to many issues. All these issues have been addressed in this part. The goals of software testing, the mindset required to perform testing, clear-cut definitions of testing terminology, phases of STLC, development of testing methodology, verification and validation, etc. have been discussed in this part.

This part will make ground for the following concepts:

- Effective testing, not exhaustive testing.
- Software testing is an established process.
- Testing should be done with the intention of finding more and more bugs, not hiding them.
- Difference between error, fault, and failure.
- Bug classification.
- Development of software testing methodology.
- Testing life cycle models.
- Difference between verification and validation.
- How to perform verification and validation at various stages of SDLC.

Chapter

1

Introduction to Software Testing

1.1 Introduction

OBJECTIVES

After reading this chapter, you should be able to understand:

- How software testing has evolved over the years
- Myths and facts of software testing
- Software testing is a separate discipline
- Testing is a complete process
- Goals of software testing
- Testing is based on a negative/destructive view
- Model for testing process
- Complete testing is not possible
- Various schools of software testing

Software has pervaded our society, from modern households to spacecrafts. It has become an essential component of any electronic device or system. This is why software development has turned out to be an exciting career for computer engineers in the last 10–15 years. However, software development faces many challenges. Software is becoming complex, but the demand for quality in software products has increased. This rise in customer awareness for quality increases the workload and responsibility of the software development team. That is why software testing has gained so much popularity in the last decade. Job trends have shifted from development to software testing. Today, software quality assurance and software testing courses are offered by many institutions. Organizations have separate testing groups with proper hierarchy. Software development is driven with testing outputs. If the testing team claims the presence of bugs in the software, then the development team cannot release the product.

However, there still is a gap between academia and the demand of industries. The practical demand is that passing graduates must be aware of testing terminologies, standards, and techniques. But students are not aware in most cases, as our universities and colleges do not offer separate software quality and testing courses. They study only software engineering. It can be said that software engineering is a mature discipline today in industry as well as in academia. On the other hand, software testing is mature in industry but not in academia. Thus, this gap must be bridged with separate courses on software quality and testing so that students do not face problems when they go for testing in industries. Today, the ideas and techniques of software testing have become essential knowledge for software developers, testers, and students as well. This book is a step forward to bridge this gap.

We cannot say that the industry is working smoothly, as far as software testing is concerned. While many industries have adopted effective software testing techniques and the development is driven by testing efforts, there are still some loopholes. Industries are dependent on automation of test execution. Therefore, testers also rely on efficient tools. But there may be an instance where automation will not help, which is why they also need to design test cases and execute them manually. Are the testers prepared for this case? This requires testing teams to have a knowledge of testing tactics and procedures of how to design test cases. This book discusses various techniques and demonstrates how to design test cases.

How do industries measure their testing process? Since software testing is a complete process today, it must be measured to check whether the process is suitable for projects. CMM (Capability Maturity Model) has measured the development process on a scale of 1–5 and companies are running for the highest scale. On the same pattern, there should be a measurement program for testing processes. Fortunately, the measurement technique for testing processes has also been developed. But how many managers, developers, testers, and of course students, know that we have a Testing Maturity Model (TMM) for measuring the maturity status of a testing process? This book gives an overview of various test process maturity models and emphasizes the need for these.

Summarizing the above discussion, it is evident that industry and academia should go parallel. Industries constantly aspire for high standards. Our university courses will have no value if their syllabi are not revised vis-à-vis industry requirements. Therefore, software testing should be included as a separate course in our curricula. On the other side, organizations cannot run with the development team looking after every stage, right from requirement gathering to implementation. Testing is an important segment of software development and it has to be thoroughly done. Therefore, there should be a separate testing group with divided responsibilities among the members.

In this chapter, we will trace the evolution of software testing. Once considered as a debugging process, it has now evolved into a complete process. Now we have software testing goals in place to have a clear picture as to why we want to study testing and execute test cases. There has been a misconception right from the evolution of software testing that it can be performed completely. But with time, we have grown out of this view and started focusing on effective testing rather than exhaustive testing. The psychology of a tester plays an important role in software testing. It matters whether one wants to show the absence of errors or their presence in the software. All these issues along with the model of testing, testing process, development of schools of testing, etc. have been discussed. This chapter presents an overview of effective software testing and its related concepts.

1.2 Evolution of Software Testing

In the early days of software development, software testing was considered only a debugging process for removing errors after the development of software. By 1970, the term 'software engineering' was in common use. But software testing was just a beginning at that time. In 1978, G. J. Myers realized the need to discuss the techniques of software testing in a separate subject. He wrote the book *The Art of Software Testing* [2] which is a classic work on software testing. He emphasized that there is a requirement that undergraduate students must learn software testing techniques so that they pass out with the basic knowledge of software testing and do not face problems in the industry. Moreover, Myers discussed the psychology of testing and emphasized that testing should be done with a mindset of finding errors and not to demonstrate that errors are not present.

By 1980, software professionals and organizations started emphasizing on quality. Organizations realized the importance of having quality assurance teams to take care of all testing activities for the project right from the beginning. In the 1990s, testing tools finally came into their own. There was a flood of various tools, which are absolutely vital to adequate testing of software systems. However, they do not solve all problems and cannot replace a testing process.

Gelperin and Hetzel [79] have characterized the growth of software testing with time. Based on this, we can divide the evolution of software testing into the following phases [80] (see Fig. 1.1).

Figure 1.1 Evolution phases of software testing

Debugging-oriented Phase (Before 1957)

This phase is the early period of testing. At that time, testing basics were unknown. Programs were written and then tested by the programmers until they were sure that all the bugs were removed. The term used for testing was *checkout*, focused on getting the system to run. Debugging was a more general term at that time and it was not distinguishable from software testing. Till 1956, there was no clear distinction between software development, testing, and debugging.

Demonstration-oriented Phase (1957–78)

The term 'debugging' continued in this phase. However, in 1957, Charles Baker pointed out that the purpose of checkout is not only to run the software but also to demonstrate the correctness according to the mentioned requirements. Thus, the scope of checkout of a program increased from program runs to program correctness. Moreover, the purpose of checkout was to show the absence of errors. There was no stress on the test case design. In this phase, there was a misconception that the software could be tested exhaustively.

Destruction-oriented Phase (1979–82)

This phase can be described as the revolutionary turning point in the history of software testing. Myers changed the view of testing from 'testing is to show the absence of errors' to 'testing is to find more and more errors.' He separated debugging from testing and stressed on the valuable test cases if they explore more bugs. This phase has given importance to effective testing in comparison to exhaustive testing. The importance of early testing was also realized in this phase.

Evaluation-oriented Phase (1983–87)

With the concept of early testing, it was realized that if the bugs were identified at an early stage of development, it was cheaper to debug them as compared to the bugs found in implementation or post-implementation phases. This phase stresses on the quality of software products such that it can be evaluated at every stage of development. In fact, the early testing concept was established in the form of verification and validation activities which help in producing better quality software. In 1983, guidelines by the National Bureau of Standards were released to choose a set of verification and validation techniques and evaluate the software at each step of software development.

Prevention-oriented Phase (1988–95)

The evaluation model stressed on the concept of bug-prevention as compared to the earlier concept of bug-detection. With the idea of early detection of bugs in earlier phases, we can prevent the bugs in implementation or further phases. Beyond this, bugs can also be prevented in other projects with the experience gained in similar software projects. The prevention model includes test planning, test analysis, and test design activities playing a major role, while the evaluation model mainly relies on analysis and reviewing techniques other than testing.

Process-oriented Phase (1996 onwards)

In this phase, testing was established as a complete process rather than a single phase (performed after coding) in the software development life cycle (SDLC). The testing process starts as soon as the requirements for a project are specified and it runs parallel to SDLC. Moreover, the model for measuring the performance of a testing process has also been developed like CMM. The model for measuring the testing process is known as Testing Maturity Model (TMM). Thus, the emphasis in this phase is also on quantification of various parameters which decide the performance of a testing process.

The evolution of software testing was also discussed by Hung Q. Nguyen and Rob Pirozzi in a white paper [81], in three phases, namely Software Testing 1.0, Software Testing 2.0, and Software Testing 3.0. These three phases discuss the evolution in the earlier phases that we described. According to this classification, the current state-of-practice is Software Testing 3.0. These phases are discussed below.

Software Testing 1.0

In this phase, software testing was just considered a single phase to be performed after coding of the software in SDLC. No test organization was there. A few testing tools were present but their use was limited due to high cost. Management was not concerned with testing, as there was no quality goal.

Software Testing 2.0

In this phase, software testing gained importance in SDLC and the concept of early testing also started. Testing was evolving in the direction of planning the test resources. Many testing tools were also available in this phase.

Software Testing 3.0

In this phase, software testing is being evolved in the form of a process which is based on strategic effort. It means that there should be a process which gives us a roadmap of the overall testing process. Moreover, it should be driven by quality goals so that all controlling and monitoring activities can be performed by the managers. Thus, the management is actively involved in this phase.

1.3 Software Testing—Myths and Facts

Before getting into the details of software testing, let us discuss some myths surrounding it. These myths are there, as this field is in its growing phase.

Myth *Testing is a single phase in SDLC*.

Truth It is a myth, at least in the academia, that software testing is just a phase in SDLC and we perform testing only when the running code of the module is ready. But in reality, testing starts as soon as we get the requirement specifications for the software. And the testing work continues throughout the SDLC, even post-implementation of the software.

Myth *Testing is easy*.

Truth This myth is more in the minds of students who have just passed out or are going to pass out of college and want to start a career in testing. So the general perception is that, software testing is an easy job, wherein test cases are executed with testing tools only. But in reality, tools are there to automate the tasks and not to carry out all testing activities. Testers' job is not easy, as they have to plan and develop the test cases manually and it requires a thorough understanding of the project being developed with its overall design. Overall, testers have to shoulder a lot of responsibility which sometimes make their job even harder than that of a developer.

Myth *Software development is worth more than testing.*

Truth This myth prevails in the minds of every team member and even in freshers who are seeking jobs. As a fresher, we dream of a job as a developer. We get into the organization as a developer and feel superior to other team members. At the managerial level also, we feel happy about the achievements of the developers but not of the testers who work towards the quality of the product being developed. Thus, we have this myth right from the beginning of our career, and testing is considered a secondary job. But testing has now

become an established path for job-seekers. Testing is a complete process like development, so the testing team enjoys equal status and importance as the development team.

Myth *Complete testing is possible.*

Truth This myth also exists at various levels of the development team. Almost every person who has not experienced the process of designing and executing the test cases manually feels that complete testing is possible. Complete testing at the surface level assumes that if we are giving all the inputs to the software, then it must be tested for all of them. But in reality, it is not possible to provide all the possible inputs to test the software, as the input domain of even a small program is too large to test. Moreover, there are many things which cannot be tested completely, as it may take years to do so. This will be demonstrated soon in this chapter. This is the reason why the term 'complete testing' has been replaced with 'effective testing.' Effective testing is to select and run some select test cases such that severe bugs are uncovered first.

Myth *Testing starts after program development.*

Truth Most of the team members, who are not aware of testing as a process, still feel that testing cannot commence before coding. But this is not true. As mentioned earlier, the work of a tester begins as soon as we get the specifications. The tester performs testing at the end of every phase of SDLC in the form of *verification* (discussed later) and plans for the *validation testing* (discussed later). He writes detailed test cases, executes the test cases, reports the test results, etc. Testing after coding is just a part of all the testing activities.

Myth *The purpose of testing is to check the functionality of the software.*

Truth Today, all the testing activities are driven by quality goals. Ultimately, the goal of testing is also to ensure quality of the software. But quality does not imply checking only the functionalities of all the modules. There are various things related to quality of the software, for which test cases must be executed.

Myth *Anyone can be a tester.*

Truth This is the extension of the myth that 'testing is easy.' Most of us think that testing is an intuitive process and it can be performed easily without any training. And therefore, anyone can be a tester. As an established process, software testing as a career also needs training for various purposes, such as to understand (i) various phases of software testing life cycle, (ii) recent techniques to design test cases, (iii) various tools and how to work on them, etc. This is the reason that various testing courses for certified testers are being run.

After having discussed the myths, we will now identify the requirements for software testing. Owing to the importance of software testing, let us first identify the concerns related to it. The next section discusses the goals of software testing.

1.4 Goals of Software Testing

To understand the new concepts of software testing and to define it thoroughly, let us first discuss the goals that we want to achieve from testing. The goals of software testing may be classified into three major categories, as shown in Fig. 1.2.

Figure 1.2 Software testing goals

Short-term or immediate goals These goals are the immediate results after performing testing. These goals may be set in the individual phases of SDLC. Some of them are discussed below.

Bug discovery The immediate goal of testing is to find errors at any stage of software development. More the bugs discovered at an early stage, better will be the success rate of software testing.

Bug prevention It is the consequent action of bug discovery. From the behaviour and interpretation of bugs discovered, everyone in the software development team gets to learn how to code safely such that the bugs discovered should not be repeated in later stages or future projects. Though errors cannot be prevented to zero, they can be minimized. In this sense, bug prevention is a superior goal of testing.

Long-term goals These goals affect the product quality in the long run, when one cycle of the SDLC is over. Some of them are discussed here.

Quality Since software is also a product, its quality is primary from the users' point of view. Thorough testing ensures superior quality. Therefore, the first goal of understanding and performing the testing process is to enhance the quality of the software product. Though quality depends on various factors, such as correctness, integrity, efficiency, etc., reliability is the major factor to achieve quality. The software should be passed through a rigorous reliability analysis to attain high quality standards. Reliability is a matter of confidence that the software will not fail, and this level of confidence increases with rigorous testing. The confidence in reliability, in turn, increases the quality, as shown in Fig. 1.3.

Figure 1.3 Testing produces reliability and quality

Customer satisfaction From the users' perspective, the prime concern of testing is customer satisfaction only. If we want the customer to be satisfied with the software product, then testing should be complete and thorough. Testing should be complete in the sense that it must satisfy the user for all the specified requirements mentioned in the user manual, as well as for the unspecified requirements which are otherwise understood. A complete testing process achieves reliability, reliability enhances the quality, and quality in turn, increases the customer satisfaction, as shown in Fig. 1.4.

Figure 1.4 Quality leads to customer satisfaction

Risk management Risk is the probability that undesirable events will occur in a system. These undesirable events will prevent the organization from successfully implementing its business initiatives. Thus, risk is basically concerned with the business perspective of an organization.

Risks must be controlled to manage them with ease. Software testing may act as a control, which can help in eliminating or minimizing risks (see Fig. 1.5).

Thus, managers depend on software testing to assist them in controlling their business goals. The purpose of software testing as a control is to provide information to management so that they can better react to risk situations [4]. For example, testing may indicate that the software being developed cannot be delivered on time, or there is a probability that high priority bugs will not be resolved by the specified time. With this advance information, decisions can be made to minimize risk situation.

Hence, it is the testers' responsibility to evaluate business risks (such as cost, time, resources, and critical features of the system being developed) and make the same a basis for testing choices. Testers should also categorize the levels of risks after their assessment (like high-risk, moderate-risk, low-risk) and this analysis becomes the basis for testing activities. Thus, risk management becomes the long-term goal for software testing.

Figure 1.5 Testing controlled by risk factors

Post-implementation goals These goals are important after the product is released. Some of them are discussed here.

Reduced maintenance cost The maintenance cost of any software product is not its physical cost, as the software does not wear out. The only maintenance cost in a software product is its failure due to errors. Post-release errors are costlier to fix, as they are difficult to detect. Thus, if testing has been done rigorously and effectively, then the chances of failure are minimized and in turn, the maintenance cost is reduced.

Improved software testing process A testing process for one project may not be successful and there may be scope for improvement. Therefore, the bug history and post-implementation results can be analysed to find out snags in the present testing process, which can be rectified in future projects. Thus, the long-term post-implementation goal is to improve the testing process for future projects.

1.5 Psychology for Software Testing

Software testing is directly related to human psychology. Though software testing has not been defined till now, but most frequently, it is defined as,

Testing is the process of demonstrating that there are no errors.

The purpose of testing is to show that the software performs its intended functions correctly. This definition is correct, but partially. If testing is performed keeping this goal in mind, then we cannot achieve the desired goals (described above in the previous section), as we will not be able to test the software as a whole. Myers first identified this approach of testing the software. This approach is based on the human psychology that human beings tend to work according to the goals fixed in their minds. If we have a preconceived assumption that the software is error-free, then consequently, we will design the test cases to show that all the modules run smoothly. But it may hide some bugs. On the other hand, if our goal is to demonstrate that a program has errors, then we will design test cases having a higher probability to uncover bugs.

Thus, if the process of testing is reversed, such that we always presume the presence of bugs in the software, then this psychology of being always suspicious of bugs widens the domain of testing. It means, now we don't think of testing only those features or specifications which have been mentioned in documents like SRS (software requirement specification), but we also think in terms of finding bugs in the domain or features which are understood but not specified. You can argue that, being suspicious about bugs in the software is a negative approach. But, this negative approach is for the benefit of constructive and effective testing. Thus, software testing may be defined as,

Testing is the process of executing a program with the intent of finding errors.

This definition has implications on the psychology of developers. It is very common that they feel embarrassed or guilty when someone finds errors in their software. However, we should not forget that humans are prone to error. We should not feel guilty for our errors. This psychology factor brings the concept that we should concentrate on discovering and preventing the errors and not feel guilt about them. Therefore, testing cannot be a joyous event unless you cast out your guilt.

According to this psychology of testing, a successful test is that which finds errors. This can be understood with the analogy of medical diagnostics of a patient. If the laboratory tests do not locate the problem, then it cannot be regarded as a successful test. On the other hand, if the laboratory test determines the disease, then the doctor can start an appropriate treatment. Thus, in

the destructive approach of software testing, the definitions of successful and unsuccessful testing should also be modified.

1.6 Software Testing Definitions

Many practitioners and researchers have defined software testing in their own way. Some are given below.

Testing is the process of executing a program with the intent of finding errors.

Myers [2]

A successful test is one that uncovers an as-yet-undiscovered error.

Myers [2]

Testing can show the presence of bugs but never their absence.

W. Dijkstra [125]

Program testing is a rapidly maturing area within software engineering that is receiving increasing notice both by computer science theoreticians and practitioners. Its general aim is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances.

E. Miller[84]

Testing is a support function that helps developers look good by finding their mistakes before anyone else does.

James Bach [83]

Software testing is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to operate.

Cem Kaner [85]

The underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large-scale and small-scale systems.

Miller [126]

Testing is a concurrent lifecycle process of engineering, using and maintaining testware (i.e. testing artifacts) in order to measure and improve the quality of the software being tested.

Craig [117]

Since quality is the prime goal of testing and it is necessary to meet the defined quality standards, software testing should be defined keeping in view the quality assurance terms. Here, it should not be misunderstood that the testing team is responsible for quality assurance. But the testing team must

be well aware of the quality goals of the software so that they work towards achieving them.

Moreover, testers these days are aware of the definition that testing is to find more and more bugs. But the problem is that there are too many bugs to fix. Therefore, the recent emphasis is on categorizing the more important bugs first. Thus, software testing can be defined as,

Software testing is a process that detects important bugs with the objective of having better quality software.

1.7 Model for Software Testing

Testing is not an intuitive activity, rather it should be learnt as a process. Therefore, testing should be performed in a planned way. For the planned execution of a testing process, we need to consider every element and every aspect related to software testing. Thus, in the testing model, we consider the related elements and team members involved (see Fig. 1.6).

Figure 1.6 Software testing model

The software is basically a part of a system for which it is being developed. Systems consist of hardware and software to make the product run. The developer develops the software in the prescribed system environment considering the testability of the software. Testability is a major issue for the developer while developing the software, as a badly written software may be difficult to test. Testers are supposed to get on with their tasks as soon as the requirements are specified. Testers work on the basis of a bug model which classifies the bugs based on the criticality or the SDLC phase in which the testing is to be performed. Based on the software type and the bug model, testers decide a testing methodology which guides how the testing will be performed. With suitable testing techniques decided in the testing methodology, testing is performed on the software with a particular goal. If the testing results are

in line with the desired goals, then the testing is successful; otherwise, the software or the bug model or the testing methodology has to be modified so that the desired results are achieved. The following describe the testing model.

Software and Software Model

Software is built after analysing the system in the environment. It is a complex entity which deals with environment, logic, programmer psychology, etc. But a complex software makes it very difficult to test. Since in this model of testing, our aim is to concentrate on the testing process, therefore the software under consideration should not be so complex such that it would not be tested. In fact, this is the point of consideration for developers who design the software. They should design and code the software such that it is testable at every point. Thus, the software to be tested may be modeled such that it is testable, avoiding unnecessary complexities.

Bug Model

Bug model provides a perception of the kind of bugs expected. Considering the nature of all types of bugs, a bug model can be prepared that may help in deciding a testing strategy. However, every type of bug cannot be predicted. Therefore, if we get incorrect results, the bug model needs to be modified.

Testing methodology and Testing

Based on the inputs from the software model and the bug model, testers can develop a testing methodology that incorporates both testing strategy and testing tactics. Testing strategy is the roadmap that gives us well-defined steps for the overall testing process. It prepares the planned steps based on the risk factors and the testing phase. Once the planned steps of the testing process are prepared, software testing techniques and testing tools can be applied within these steps. Thus, testing is performed on this methodology. However, if we don't get the required results, the testing plans must be checked and modified accordingly.

All the components described above will be discussed in detail in subsequent chapters.

1.8 Effective Software Testing vs. Exhaustive Software Testing

Exhaustive or complete software testing means that every statement in the program and every possible path combination with every possible combination of data must be executed. But soon, we will realize that exhaustive testing is out of scope. That is why the questions arise: (i) When are we done with testing? or (ii) How do we know that we have tested enough? There may be

many answers for these questions with respect to time, cost, customer, quality, etc. This section will explore that exhaustive or complete testing is not possible. Therefore, we should concentrate on effective testing which emphasizes efficient techniques to test the software so that important features will be tested within the constrained resources.

The testing process should be understood as a domain of possible tests (see Fig. 1.7). There are subsets of these possible tests. But the domain of possible tests becomes infinite, as we cannot test every possible combination.

Figure 1.7 Testing domain

This combination of possible tests is infinite in the sense that the processing resources and time are not sufficient for performing these tests. Computer speed and time constraints limit the possibility of performing all the tests. Complete testing requires the organization to invest a long time which is not cost-effective. Therefore, testing must be performed on selected subsets that can be performed within the constrained resources. This selected group of subsets, but not the whole domain of testing, makes effective software testing. Effective testing can be enhanced if subsets are selected based on the factors which are required in a particular environment.

Now, let us see in detail why complete testing is not possible.

The Domain of Possible Inputs to the Software is too Large to Test

If we consider the input data as the only part of the domain of testing, even then, we are not able to test the complete input data combination. The domain of input data has four sub-parts: (a) valid inputs, (b) invalid inputs, (c) edited inputs, and (d) race condition inputs (See Fig. 1.8)

Valid inputs It seems that we can test every valid input on the software. But look at a very simple example of adding two-digit two numbers. Their range is from –99 to 99 (total 199). So the total number of test case combinations will be $199 \times 199 = 39601$. Further, if we increase the range from two digits to four-digits, then the number of test cases will be 399,960,001. Most addition programs accept 8 or 10 digit numbers or more. How can we test all these combinations of valid inputs?

Invalid inputs Testing the software with valid inputs is only one part of the input sub-domain. There is another part, invalid inputs, which must be

tested for testing the software effectively. The important thing in this case is the behaviour of the program as to how it responds when a user feeds invalid inputs. The set of invalid inputs is also too large to test. If we consider again the example of adding two numbers, then the following possibilities may occur from invalid inputs:

(i) Numbers out of range
(ii) Combination of alphabets and digits
(iii) Combination of all alphabets
(iv) Combination of control characters
(v) Combination of any other key on the keyboard

Figure 1.8 Input domain for testing

Edited inputs If we can edit inputs at the time of providing inputs to the program, then many unexpected input events may occur. For example, you can add many spaces in the input, which are not visible to the user. It can be a reason for non-functioning of the program. In another example, it may be possible that a user is pressing a number key, then Backspace key continuously and finally after sometime, he presses another number key and Enter. Its input buffer overflows and the system crashes.

The behaviour of users cannot be judged. They can behave in a number of ways, causing defect in testing a program. That is why edited inputs are also not tested completely.

Race condition inputs The timing variation between two or more inputs is also one of the issues that limit the testing. For example, there are two input events,

A and B. According to the design, A precedes B in most of the cases. But, B can also come first in rare and restricted conditions. This is the race condition, whenever B precedes A. Usually the program fails due to race conditions, as the possibility of preceding B in restricted condition has not been taken care, resulting in a race condition bug. In this way, there may be many race conditions in the system, especially in multiprocessing systems and interactive systems. Race conditions are among the least tested.

There are too Many Possible Paths Through the Program to Test

A program path can be traced through the code from the start of a program to its termination. Two paths differ if the program executes different statements in each, or executes the same statements but in different order. A testing person thinks that if all the possible paths of control flow through the program are executed, then possibly the program can be said to be completely tested. However, there are two flaws in this statement.

(i) The number of unique logic paths through a program is too large. This was demonstrated by Myers[2] with an example shown in Fig. 1.9. It depicts a 10–20 statements program consisting of a DO loop that iterates up to 20 times. Within the body of the DO loop is a set of nested IF statements. The number of all the paths from point A to B is approximately 10^{14}. Thus, all these paths cannot be tested, as it may take years of time.

Figure 1.9 Sample flow graph 1

See another example for the code fragment shown in Fig. 1.10 and its corresponding flow graph in Fig. 1.11 (We will learn how to convert the program into a flow graph in Chapter 5).

```
for (int i = 0; i < n; ++i)
{
if (m >=0)
     x[i] = x[i] + 10;
else
     x[i] = x[i] - 2;
}
```

Figure 1.10 Sample code fragment

Figure 1.11 Example flow graph 2

Now calculate the number of paths in this fragment. For calculating the number of paths, we must know how many paths are possible in one iteration. Here in our example, there are two paths in one iteration. Now the total number of paths will be $2^n + 1$, where n is the number of times the loop will be carried out, and 1 is added, as the `for` loop will exit after its looping ends and it terminates. Thus, if n is 20, then the number of paths will be $2^{20} + 1$, i.e. 1048577. Therefore, all these paths cannot be tested, as it may take years.

(ii) The complete path testing, if performed somehow, does not guarantee that there will *not* be errors. For example, it does not claim that a program matches its specification. If one were asked to write an ascending order sorting program but the developer mistakenly produces a descending order program, then exhaustive path testing will be of little value. In another case, a program may be incorrect because of missing paths. In this case, exhaustive path testing would not detect the missing path.

Every Design Error Cannot be Found

Manna and Waldinger [15] have mentioned the following fact: 'We can never be sure that the specifications are correct.' How do we know that the specifications are achievable? Its consistency and completeness must be proved, and in general, that is a provably unsolvable problem [9]. Therefore, specification errors are one of the major reasons that make the design of the software faulty. If the user requirement is to have measurement units in inches and the specification says that these are in meters, then the design will also be in meters. Secondly, many user interface failures are also design errors.

The study of these limitations of testing shows that the domain of testing is infinite and testing the whole domain is just impractical. When we leave a single test case, the concept of complete testing is abandoned. But it does not mean that we should not focus on testing. Rather, we should shift our attention from exhaustive testing to effective testing. Effective testing provides the flexibility to select only the subsets of the domain of testing based on project priority such that the chances of failure in a particular environment is minimized.

1.9 Effective Testing is hard

We have seen the limitations of exhaustive software testing which makes it nearly impossible to achieve. Effective testing, though not impossible, is hard to implement. But if there is careful planning, keeping in view all the factors which can affect it, then it is implementable as well as effective. To achieve that planning, we must understand the factors which make effective testing difficult. At the same time, these factors must be resolved. These are described as follows.

Defects are hard to find The major factor in implementing effective software testing is that many defects go undetected due to many reasons, e.g. certain test conditions are never tested. Secondly, developers become so familiar with their developed system that they overlook details and leave some parts untested. So a proper planning for testing all the conditions should be done and independent testing, other than that done by developers, should be encouraged.

When are we done with testing This factor actually searches for the definition of effective software testing. Since exhaustive testing is not possible, we don't know what should be the criteria to stop the testing process. A software engineer needs more rigorous criteria for determining when sufficient testing has been performed. Moreover, effective testing has the limiting factor of cost, time, and personnel. In a nutshell, the criteria should be developed for enough

testing. For example, features can be prioritized which must be tested within the boundary of cost, time, and personnel of the project.

1.10 Software Testing as a Process

Since software development is an engineering activity for a quality product, it consists of many processes. As it was seen in testing goals, software quality is the major driving force behind testing. Software testing has also emerged as a complete process in software engineering (see Fig. 1.12). Therefore, our major concern in this text is to show that testing is not just a phase in SDLC normally performed after coding, rather software testing is a process which runs parallel to SDLC. In Fig. 1.13, you can see that software testing starts as soon as the requirements are specified. Once the SRS document is prepared, testing process starts. Some examples of test processes, such as test plan, test design, etc. are given. All the phases of testing life cycle will be discussed in detail in the next chapter.

Figure 1.12 Testing process emerged out of development process

Figure 1.13 Testing process runs parallel to software process

Software testing process must be planned, specified, designed, implemented, and quantified. Testing must be governed by the quality attributes of the software product. Thus, testing is a dual-purpose process, as it is used to detect bugs as well as to establish confidence in the quality of software.

An organization, to ensure better quality software, must adopt a testing process and consider the following points:

- Testing process should be organized such that there is enough time for important and critical features of the software.

- Testing techniques should be adopted such that these techniques detect maximum bugs.
- Quality factors should be quantified so that there is a clear understanding in running the testing process. In other words, the process should be driven by quantified quality goals. In this way, the process can be monitored and measured.
- Testing procedures and steps must be defined and documented.
- There must be scope for continuous process improvement.

All the issues related to testing process will be discussed in succeeding chapters.

1.11 Schools of Software Testing

Software testing has also been classified into some views according to some practitioners. They call these views or ideas as *schools of testing*. The idea of schools of testing was given by Bret Pettichord [82]. He has proposed the following schools:

Analytical School of Testing

In this school of testing, software is considered as a logical artifact. Therefore, software testing techniques must have a logico-mathematical form. This school requires that there must be precise and detailed specifications for testing the software. Moreover, it provides an objective measure of testing. After this, testers are able to verify whether the software conforms to its specifications. Structural testing is one example for this school of testing. Thus, the emphasis is on testing techniques which should be adopted.

This school defines software testing as a branch of computer science and mathematics.

Standard School of Testing

The core beliefs of this school of testing are:

1. Testing must be managed (for example, through traceability matrix. It will be discussed in detail in succeeding chapters). It means the testing process should be predictable, repeatable, and planned.
2. Testing must be cost-effective.
3. Low-skilled workers require direction.
4. Testing validates the product.
5. Testing measures development progress.

Thus, the emphasis is on measurement of testing activities to track the development progress.

This school defines software testing as a managed process.

The implications of this school are:

1. There must be clear boundaries between testing and other activities.
2. Plans should not be changed as it complicates progress tracking.
3. Software testing is a complete process.
4. There must be some test standards, best practices, and certification.

Quality School of Testing

The core beliefs of this school of testing are:

1. Software quality requires discipline.
2. Testing determines whether development processes are being followed.
3. Testers may need to police developers to follow the rules.
4. Testers have to protect the users from bad software.

Thus, the emphasis is to follow a good process.

This school defines software testing as a branch of software quality assurance.

The implications of this school are:

1. It prefers the term 'quality assurance' over 'testing.'
2. Testing is a stepping stone to 'process improvement.'

Context-driven School of Testing

This school is based on the concept that testing should be performed according to the context of the environment and project. Testing solutions cannot be the same for every context. For example, if there is a high-cost real-time defense project, then its testing plans must be different as compared to any daily-life low-cost project. Test plan issues will be different for both projects. Therefore, testing activities should be planned, designed, and executed keeping in view the context of environment in which testing is to be performed. The emphasis is to select a testing type that is valuable. Thus, context-driven testing can be defined as the testing driven by environment, type of project, and the intended use of software.

The implications of this school are:

1. Expect changes. Adapt testing plans based on test results.
2. Effectiveness of test strategies can only be determined with field research.

3. Testing research requires empirical and psychological study.
4. Focus on skill over practice.

Agile School of Testing

This type of school is based on testing the software which is being developed by iterative method of development and delivery. In this type of process model, the software is delivered in a short span of time; and based on the feedback, more features and capabilities are added. The focus is on satisfying the customer by delivering a working software quickly with minimum features and then improvising on it based on the feedback. The customer is closely related to the design and development of the software. Since the delivery timelines are short and new versions are built by modifying the previous one, chances of introducing bugs are high during the changes done to one version. Thus, regression testing becomes important for this software. Moreover, test automation also assumes importance to ensure the coverage of testing in a short span of time.

It can be seen that agile software development faces various challenges. This school emphasizes on all the issues related to agile testing.

1.12 Software Failure Case Studies

At the end of this chapter, let us discuss a few case studies that highlight the failures of some expensive and critical software projects. These case studies show the importance of software testing. Many big projects have failed in the past due to lack of proper software testing. In some instances, the product was replaced without question. The concerned parties had to bear huge losses in every case. It goes on to establish the fact that the project cost increases manifold if a product is launched without proper tests being performed on it. These case studies emphasize the importance of planning the tests, designing, and executing the test cases in a highly prioritized way, which is the central theme of this book.

Air Traffic Control System Failure (September 2004)

In September 2004, air traffic controllers in the Los Angeles area lost voice contact with 800 planes allowing 10 to fly too close together, after a radio system shut down. The planes were supposed to be separated by five nautical miles laterally, or 2,000 feet in altitude. But the system shut down while 800 planes were in the air, and forced delays for 400 flights and the cancellations of 600 more. The system had voice switching and control system, which gives controllers a touch-screen to connect with planes in flight and with controllers across the room or in distant cities.

The reason for failure was partly due to a 'design anomaly' in the way Microsoft Windows servers were integrated into the system. The servers were timed to shut down after 49.7 days of use in order to prevent a data overload. To avoid this automatic shutdown, technicians are required to restart the system manually every 30 days. An improperly trained employee failed to reset the system, leading it to shut down without warning.

Welfare Management System Failure (July 2004)

It was a new government system in Canada costing several hundred million dollars. It failed due to the inability to handle a simple benefits rate increase after being put into live operation. The system was not given adequate time for system and acceptance testing and never tested for its ability to handle a rate increase.

Northeast Blackout (August 2003)

It was the worst power system failure in North American history. The failure involved loss of electrical power to 50 million customers, forced shutdown of 100 power plants and economic losses estimated at $6 billion. The bug was reportedly in one utility company's vendor-supplied power monitoring and management system. The failures occurred when multiple systems trying to access the same information at once got the equivalent of busy signals. The software should have given one system precedent. The error was found and corrected after examining millions of lines of code.

Tax System Failure (March 2002)

This system was Britain's national tax system which failed in 2002 and resulted in more than 100,000 erroneous tax overcharges. It was suggested in the error report that the integration testing of multiple parts could not be done.

Mars Polar Lander Failure (December 1999)

NASA's Mars Polar Lander was to explore a unique region of the red planet; the main focus was on climate and water. The spacecraft was outfitted with a robot arm which was capable of digging into Mars in search for near-surface ice. It was supposed to gently set itself down near the border of Mars' southern polar cap. But it couldn't be successful to touch the surface of Mars. The communication was lost when it was 1800 meters away from the surface of Mars.

When the Lander's legs started opening for landing on Martian surface, there were vibrations which were identified by the software. This resulted in the vehicle's descent engines being cut off while it was still 40 meters above the surface, rather than on touchdown as planned. The software design failed to take into account that a touchdown signal could be detected before the

Lander actually touched down. The error was in design. It should have been configured to disregard touchdown signals during the deployment of the Lander's legs.

Mars Climate Orbiter Failure (September 1999)

Mars Climate Orbiter was one of a series of missions in a long-term program of Mars exploration managed by the Jet Propulsion Laboratory for NASA's Office of Space Science, Washington, DC. Mars Climate Orbiter was to serve as a communications relay for the Mars Polar Lander mission. But it disappeared as it began to orbit Mars. Its cost was about $125 million. The failure was due to an error in transfer of information between a team in Colorado and a team in California. This information was critical to the maneuvers required to place the spacecraft in the proper Mars orbit. One team used English units (e.g. inches, feet, and pounds), while the other team used metric units for a key spacecraft operation.

Stock Trading Service Failure (February 1999)

This was an online US stock trading service which failed during trading hours several times over a period of days in February 1999. The problem found was due to bugs in a software upgrade intended to speed online trade confirmations.

Intel Pentium Bug (April 1997)

Intel Pentium was also observed with a bug that is known as Dan-0411 or Flag Erratum. The bug is related to the operation where conversion of floating point numbers is done into integer numbers. All floating-point numbers are stored inside the microprocessor in an 80-bit format. Integer numbers are stored externally in two different sizes, i.e. 16 bits for short integers and 32 bits for long integers. It is often desirable to store the floating-point numbers as integer numbers. When the converted numbers won't fit the integer size range, a specific error flag is supposed to be set in a floating point status register. But the Pentium II and Pentium Pro fail to set this error flag in many cases.

The Explosion of Ariane 5 (June 1996)

Ariane 5 was a rocket launched by the European Space Agency. On 4 June 1996, it exploded at an altitude of about 3700 meters just 40 seconds after its lift-off from Kourou, French Guiana. The launcher turned off its flight path, broke up and exploded. The rocket took a decade of development time with a cost of $7 billion. The destroyed rocket and its cargo were valued at $500 million. The failure of Ariane was caused due to the complete loss of guidance and altitude information, 37 seconds after the start of main engine ignition sequence (30 seconds after lift-off).

A board of inquiry investigated the causes of the explosion and in two weeks issued a report. It was found that the cause of the failure was a software error in the inertial reference system (SRI). The internal SRI software exception was caused during the execution of a data conversion from 64-bit floating point to 16-bit signed integer value. A 64-bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16-bit signed integer. The number was larger than 32,767, the largest integer stored in a 16-bit signed integer; and thus the conversion failed. The error was due to specification and design errors in the software of the inertial reference system.

SUMMARY

This chapter emphasizes that software testing has emerged as a separate discipline. Software testing is now an established process. It is driven largely by the quality goals of the software. Thus, testing is the critical element of software quality. This chapter shows that testing cannot be performed with an optimistic view that the software does not contain errors. Rather, testing should be performed keeping in mind that the software always contains errors.

A misconception has prevailed through the evolution of software testing that complete testing is possible, but it is not true. Here, it has been demonstrated that complete testing is not possible. Thus, the term 'effective software testing' is becoming more popular as compared to 'exhaustive' or 'complete testing'. The chapter gives an overview of software testing discipline along with definitions of testing, model for testing, and different schools of testing. To realize the importance of effective software testing as a separate discipline, some case studies showing the software failures in systems have also been discussed.

Let us quickly review the important concepts described in this chapter.

- Software testing has evolved through many phases, namely (i) debugging-oriented phase, (ii) demonstration-oriented phase, (iii) destruction-oriented phase, (iv) evaluation-oriented phase, (v) prevention-oriented phase, and (vi) process-oriented phase.
- There is another classification for evolution of software testing, namely Software testing 1.0, Software testing 2.0, and Software testing 3.0.
- Software testing goals can be partitioned into following categories:
  1. Immediate goals
     - Bug discovery
     - Bug prevention
  2. Long-term goals
     - Reliability
     - Quality
     - Customer satisfaction
     - Risk management

3. Post-implementation goals
   - Reduced maintenance cost
   - Improved testing process

- Testing should be performed with a mindset of finding bugs. This suspicious strategy (destructive approach) helps in finding more and more bugs.
- Software testing is a process that detects important bugs with the objective of having better quality software.
- Exhaustive testing is not possible due to the following reasons:
  - It is not possible to test every possible input, as the input domain is too large.
  - There are too many possible paths through the program to test.
  - It is difficult to locate every design error.
- Effective software testing, instead of complete or exhaustive testing, is adopted such that critical test cases are covered first.
- There are different views on how to perform testing which have been categorized as schools of software testing, namely (i) analytical school, (ii) standard school, (iii) quality school, (iv) context school, and (v) agile school.
- Software testing is a complete process like software development.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Bug discovery is a _____ goal of software testing.
   (a) Long-term
   (b) Short-term
   (c) Post-implementation
   (d) All
2. Customer satisfaction and risk management are _____ goals of software testing.
   (a) Long-term
   (b) Short-term
   (c) Post-implementation
   (d) All
3. Reduced maintenance is a _____ goal of software testing.
   (a) Long-term
   (b) Short-term
   (c) Post-implementation
   (d) All

4. Software testing produces _____.
   (a) Reliability
   (b) Quality
   (c) Customer Satisfaction
   (d) All
5. Testing is the process of _____ errors.
   (a) Hiding
   (b) Finding
   (c) Removing
   (d) None
6. Complete testing is _____.
   (a) Possible
   (b) Impossible
   (c) None
7. The domain of possible inputs to the software is too _____ to test.
   (a) Large
   (b) Short
   (c) none
8. The set of invalid inputs is too _____ to test.
   (a) Large
   (b) Short
   (c) none
9. Race conditions are among the _____ tested.
   (a) Most
   (b) Least
   (c) None
10. Every design error _____ be found.
   (a) Can
   (b) Can definitely
   (c) Cannot
   (d) None

Review Questions

1. How does testing help in producing quality software?
2. 'Testing is the process of executing a program with the intent of finding errors.' Comment on this statement.
3. Differentiate between effective and exhaustive software testing.

4. Find out some myths related to software testing, other than those described in this chapter.
5. 'Every design error cannot be found.' Discuss this problem in reference to some project.
6. 'The domain of possible inputs to the software is too large to test.' Demonstrate using some example programs.
7. 'There are too many possible paths through the program to test.' Demonstrate using some example programs.
8. What are the factors for determining the limit of testing?
9. Explore some more software failure case studies other than those discussed in this chapter.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

2

Software Testing Terminology and Methodology

OBJECTIVES

After reading this chapter, you should be able to understand:

- Difference between error, fault, and failure
- Life cycle of a bug
- How a bug affects the economics of software testing
- How the bugs are classified
- Testing principles
- Software Testing Life Cycle (STLC) and its models
- Difference between verification and validation
- Development of software testing methodology

Since software testing or rather effective software testing is an emerging field, many related terms are either undefined or yet to be known. We tend to use the terms 'error', 'bug', 'defect', 'fault', 'failure', etc. interchangeably. But they don't mean the same.

Bugs in general, are more important during software testing. All the bugs do not have the same level of severity. Some bugs are less important, as they are not going to affect the product badly. On the other hand, bugs with high severity level will affect the product such that the product may not be released. Therefore, bugs need to be classified according to their severity.

A bug has a complete cycle from its initiation to death. Moreover, these bugs also affect a project cost at various levels of development. If there is a bug at the specification level and it is caught, then it is more economical to debug it as compared to the case if it is found in the implementation stage. If the bug propagates further into later stages of software development life cycle (SDLC), it would become more costly to debug.

As we have discussed in the previous chapter, software testing today is recognized as a complete process, and we need to identify various stages of software testing life cycle. These stages, like SDLC stages, define various tasks to be done during the testing of software in a hierarchy. Besides this, what should be our complete methodology to test the software? How should we go for testing a complete project? What will be the techniques used during testing? Should we use testing tools during testing? This chapter defines all the terms related to software testing, some concepts related to these terms, and then discusses the development of a testing methodology.

2.1 Software Testing Terminology

2.1.1 Definitions

As mentioned earlier, terms like error, bug, failure, defect, etc. are not synonymous and hence these should not be used interchangeably. All these terms are defined below.

Failure When the software is tested, *failure* is the first term being used. It means the inability of a system or component to perform a required function according to its specification. In other words, when results or behaviour of the system under test are different as compared to specified expectations, then failure exists.

Fault/Defect/Bug Failure is the term which is used to describe the problems in a system on the output side, as shown in Fig. 2.1. *Fault* is a condition that in actual causes a system to produce failure. Fault is synonymous with the words *defect* or *bug*. Therefore, fault is the reason embedded in any phase of SDLC and results in failures. It can be said that failures are manifestation of bugs. One failure may be due to one or more bugs and one bug may cause one or more failures. Thus, when a bug is executed, then failures are generated. But this is not always true. Some bugs are hidden in the sense that these are not executed, as they do not get the required conditions in the system. So, hidden bugs may not always produce failures. They may execute only in certain rare conditions.

Figure 2.1 Testing terminology

Error Whenever a development team member makes a mistake in any phase of SDLC, errors are produced. It might be a typographical error, a misleading of a specification, a misunderstanding of what a subroutine does, and so on. Error is a very general term used for human mistakes. Thus, an error causes a bug and the bug in turn causes failures, as shown in Fig. 2.2.

Figure 2.2 Flow of faults

Example 2.1

Consider the following module in a software:

```
Module A()
{
    ….

    while(a > n+1);
    {
     …
         print("The value of x is", x);
     }

     ….
}
```

Suppose the module shown above is expected to print the value of `x` which is critical for the use of software. But when this module will be executed, the value of `x` will not be printed. It is a *failure* of the program. When we try to look for the reason of the failure, we find that in `Module A()`, the `while` loop is not being executed. A condition is preventing the body of `while` loop to be executed. This is known as *bug/defect/fault*. On close observation, we find a semicolon being misplaced after the `while` loop which is not its correct syntax and it is not allowing the loop to execute. This mistake is known as an *error*.

Test case Test case is a well-documented procedure designed to test the functionality of a feature in the system. A test case has an identity and is associated with a program behaviour. The primary purpose of designing a test case is to find errors in the system. For designing the test case, it needs to provide a set of inputs and its corresponding expected outputs. The sample for a test case is shown in Fig. 2.3.

Test Case ID
Purpose
Preconditions
Inputs
Expected Outputs

Figure 2.3 Test case template

Test case ID is the identification number given to each test case.

Purpose defines why the case is being designed.

Preconditions for running the inputs in a system can be defined, if required, in a test case.

Inputs should not be hypothetical. Actual inputs must be provided, instead of general inputs. For example, if two integer numbers have to be provided as input, then specifically mention them as 23 and 56.

Expected outputs are the outputs which should be produced when there is no failure.

Test cases are designed based on the chosen testing techniques. They provide inputs when the system is executed. After execution, observed results are compared with expected outputs mentioned in the test case.

Testware The documents created during testing activities are known as *testware.* Taking the analogy from software and hardware as a product, testware are the documents that a test engineer produces. It may include test plans, test specifications, test case design, test reports, etc. Testware documents should also be managed and updated like a software product.

Incident When a failure occurs, it may or may not be readily apparent to the user. An *incident* is the symptom(s) associated with a failure that alerts the user about the occurrence of a failure.

Test oracle An *oracle* is the means to judge the success or failure of a test, i.e. to judge the correctness of the system for some test. The simplest oracle is comparing actual results with expected results by hand. This can be very time-consuming, so automated oracles are sought.

2.1.2 Life Cycle of a Bug

It should be clear that any member of the development team can make an error in any phase of SDLC. If an error has been produced in the requirement specification phase and not detected in the same phase, then it results in a bug in the next phase, i.e. the design phase. In the design phase, a bug has come from the previous stage, but an error can also be produced in this stage. Again, if the error in this phase is not detected and it passes on to the next stage, i.e. coding phase, then it becomes a bug. In this way, errors and bugs appear and travel through various stages of SDLC, as shown in Fig. 2.4. It means, one stage may contain errors as well as bugs and the bugs which come from the previous stage are harder to detect and debug.

In the testing phase, we analyse the incidents when the failure occurs. On the basis of symptoms derived from the incidents, a bug can be classified into certain categories. After this, the bug can be isolated in the corresponding phase of SDLC and resolved by finding its exact location.

The whole life cycle of a bug can be classified into two phases: (i) bugs-in phase and (ii) bugs-out phase.

Figure 2.4 Life cycle of a bug

Bugs-In Phase

This phase is where the errors and bugs are introduced in the software. Whenever we commit a mistake, it creates errors on a specific location of the software and consequently, when this error goes unnoticed, it causes some conditions to fail, leading to a bug in the software. This bug is carried out to the subsequent phases of SDLC, if not detected. Thus, a phase may have its own errors as well as bugs received from the previous phase. If you are not performing verification (discussed later in this chapter) on earlier phases, then there is no chance of detecting these bugs.

Bugs-Out Phase

If failures occur while testing a software product, we come to the conclusion that it is affected by bugs. However, there are situations when bugs are present, even though we don't observe any failures. That is another issue of discussion. In this phase, when we observe failures, the following activities are performed to get rid of the bugs.

Bug classification In this part, we observe the failure and classify the bugs according to its nature. A bug can be critical or catastrophic in nature or it may have no adverse effect on the output behaviour of the software. In this way, we classify all the failures. This is necessary, because there may be many bugs to be resolved. But a tester may not have sufficient time. Thus, categorization of bugs may help by handling high criticality bugs first and considering other trivial bugs on the list later, if time permits. We have taken various considerations to classify different bugs (discussed later in this chapter).

Bug isolation Bug isolation is the activity by which we locate the module in which the bug appears. Incidents observed in failures help in this activity. We observe the symptoms and back-trace the design of the software and reach the module/files and the condition inside it which has caused the bug. This is known as bug isolation.

Bug resolution Once we have isolated the bug, we back-trace the design to pinpoint the location of the error. In this way, a bug is resolved when we have found the exact location of its occurrence.

2.1.3 States of a Bug

Based on the above discussion, a bug attains the following different states in its life cycle (see Fig. 2.5).

New The state is new when the bug is reported first time by a tester.

Open The new state does not verify that the bug is genuine. When the test leader (test organization hierarchy will be discussed in Chapter 9) approves that the bug is genuine, its state becomes open.

Assign An open bug comes to the development team where the development team verifies its validity. If the bug is valid, a developer is assigned the job to fix it and the state of the bug now is 'ASSIGN'.

Deferred The developer who has been assigned to fix the bug will check its validity and priority. If the priority of the reported bug is not high or there is not sufficient time to test it or the bug does not have any adverse effect on the software, then the bug is changed to deferred state which implies the bug is expected to be fixed in next releases.

Rejected It may be possible that the developer rejects the bug after checking its validity, as it is not a genuine one.

Test After fixing the valid bug, the developer sends it back to the testing team for next round of checking. Before releasing to the testing team, the developer changes the bug's state to 'TEST'. It specifies that the bug has been fixed by the development team but not tested and is released to the testing team.

Verified/fixed The tester tests the software and verifies whether the reported bug is fixed or not. After verifying, the developer approves that the bug is fixed and changes the status to 'VERIFIED'.

Reopened If the bug is still there even after fixing it, the tester changes its status to 'REOPENED'. The bug traverses the life cycle once again. In another case, a bug which has been closed earlier may be reopened if it appears again. In this case, the status will be REOPENED instead of OPEN.

Closed Once the tester and other team members are confirmed that the bug is completely eliminated, they change its status to 'CLOSED'.

Figure 2.5 States of a bug

The states of a bug, as shown in Fig. 2.5, are in general and the terminology or sequence of hierarchy may differ in organizations.

2.1.4 Why do Bugs occur?

This is a very basic question for testers. The following points can answer this best:

To Err is Human

As discussed earlier, errors are produced when developers commit mistakes. Sample this. The phone rang while coding and the developer got distracted, he pressed a wrong key and the results of that mistake produced a bug in the system. But this is not the only reason for bugs. Consider the second point.

Bugs in Earlier Stages go Undetected and Propagate

The phases of SDLC are connected to each other and the output of one phase becomes the input to the next. Therefore, an undetected bug easily propagates into subsequent phases. These propagated bugs, in later phases, are harder to detect and if found, are costlier to debug. It is a proven fact that requirement specification and design phases contain the largest percentage of bugs.

This point also establishes the fact that testing should be performed at each level of SDLC. Errors are inevitable; however, we can prevent critical bugs from propagating into later stages by checking minute details at each level of SDLC.

How do bugs get into a software product? There may be various reasons: unclear or constantly changing requirements, software complexity, programming errors, timelines, errors in bug tracking, communication gap, documentation errors, deviation from standards, etc. Some examples are given below:

- Miscommunication in gathering requirements from the customer is a big source of bugs.
- If the requirements keep changing from time to time, it creates a lot of confusion and pressure, both on the development as well as the testing team. Often, a new feature added or an existing feature removed can be linked to other modules or components in the software.
- If the effect of changes in one module to another module is overlooked, it causes bugs.
- Rescheduling of resources, re-doing or discarding an already completed work, and changes in hardware/software requirements can affect the software. Assigning a new developer to the project midway can cause bugs. If proper coding standards have not been followed, then the new developer might not get all the relevant details of the project. Improper code documentation and ineffective knowledge transfer can limit the developer's ability to produce error-free codes. Discarding a portion of the existing code might just leave its trail behind in other parts of the software. Overlooking or not eliminating such code can cause bugs. Serious bugs can especially occur with larger projects, as it gets tougher to identify the problem area.
- Complexity in keeping a track of all the bugs can in turn cause more bugs. This gets harder when a bug has a very complex life cycle, i.e. when the number of times it has been closed, re-opened, not accepted, ignored, etc. goes on increasing.

2.1.5 Bugs Affect Economics of Software Testing

Studies have demonstrated that testing prior to coding is 50% effective in detecting errors and after coding, it is 80% effective. Moreover, it is at least 10 times as costly to correct an error after coding as before, and 100 times as

costly to correct a production error (post-release error). This is how the bugs affect the economics of testing.

There is no guarantee that all the errors will be recognized after testing and that the testing phase will remove all of them. This is not possible practically. If the bugs embedded in earlier stages go undetected, it is more difficult to detect them in later stages. So the cost to remove bugs from a system increases, as *the cost of a bug is equal to Detection Cost + Correction Cost.*

A bug found and fixed during early stages when the specification is being written, cost next to nothing. The same bug, if not found until the software is coded and tested, might cost ten times the cost in early stages. Moreover, after the release of the product, if a customer finds the same bug, the cost will be 100 times more. Figure 2.6 outlines the relative cost to correct a defect depending on the SDLC in which it is discovered. Therefore, cost increase is logarithmic.

Figure 2.6 Cost of debugging increases if bug propagates

2.1.6 Bug Classification Based on Criticality

Bugs can be classified based on the impact they have on the software under test. This classification can be used for the prioritization of bugs, as all bugs cannot be resolved in one release. Since all bugs are not of the same criticality, prioritization will put high criticality bugs on top of the list. We have divided bugs based on their criticality in the following broad categories:

Critical Bugs

This type of bugs has the worst effect such that it stops or hangs the normal functioning of the software. The person using the software becomes helpless when this type of bug appears. For example, in a sorting program, after providing the input numbers, the system hangs and needs to be reset.

Major Bug

This type of bug does not stop the functioning of the software but it causes a functionality to fail to meet its requirements as expected. For example, in a sorting program, the output is being displayed but not the correct one.

Medium Bugs

Medium bugs are less critical in nature as compared to critical and major bugs. If the outputs are not according to the standards or conventions, e.g. redundant or truncated output, then the bug is a medium bug.

Minor Bugs

These types of bugs do not affect the functionality of the software. These are just mild bugs which occur without any effect on the expected behaviour or continuity of the software. For example, typographical error or misaligned printout.

2.1.7 Bug Classification Based on SDLC

Since bugs can appear in any phase of SDLC, they can be classified based on SDLC phases which are described below [9, 48].

Requirements and Specifications Bugs

The first type of bug in SDLC is in the requirement gathering and specification phase. It has been observed that most of the bugs appear in this phase only. If these bugs go undetected, they propagate into subsequent phases. Requirement gathering and specification is a difficult phase in the sense that requirements gathered from the customer are to be converted into a *requirement specification* which will become the base for design. There may be a possibility that requirements specified are not exactly what the customers want. Moreover, specified requirements may be incomplete, ambiguous, or inconsistent. Specification problems lead to wrong missing, or superfluous features.

Design Bugs

Design bugs may be the bugs from the previous phase and in addition those errors which are introduced in the present phase. The following design errors may be there.

Control flow bugs If we look at the control flow of a program (through control flow graph–discussed in Chapter 5), then there may be many errors. For example, some paths through the flow may be missing; there may be unreachable paths, etc.

Logic bugs Any type of logical mistakes made in the design is a logical bug. For example, improper layout of cases, missing cases, improper combination of cases, misunderstanding of the semantics of the order in which a Boolean expression is evaluated.

Processing bugs Any type of computation mistakes result in processing bugs. Examples include arithmetic error, incorrect conversion from one data representation to another, ignoring overflow, improper use of logical operators, etc.

Data flow bugs Control flow cannot identify data errors. For this, we use data flow (through data flow graph–discussed in Chapter 5). There may be data flow anomaly errors like un-initialized data, initialized in wrong format, data initialized but not used, data used but not initialized, redefined without any intermediate use, etc.

Error handling bugs There may be errors about error handling in the software. There are situations in the system when exception handling mechanisms must be adopted. If the system fails, then there must be an error message or the system should handle the error in an appropriate way. If you forget to do all this, then error handling bugs appear.

Race condition bugs Race conditions (discussed in Chapter 1) also lead to bugs. Sometimes these bugs are irreproducible.

Boundary-related bugs Most of the time, the designers forget to take into consideration what will happen if any aspect of a program goes beyond its minimum and maximum values. For example, there is one integer whose range is between 1 to 100. What will happen if a user enters a value as 1 or 101? When the software fails at the boundary values, then these are known as boundary-related bugs. There may be boundaries in loop, time, memory, etc.

User interface bugs There may be some design bugs that are related to users. If the user does not feel good while using the software, then there are user interface bugs. Examples include inappropriate functionality of some features; not doing what the user expects; missing, misleading, or confusing information; wrong content in the help text; inappropriate error messages, etc.

Coding Bugs

There may be a long list of coding bugs. If you are a programmer, then you are aware of some common mistakes made. For example, undeclared data, undeclared routines, dangling code, typographical errors, documentation bugs, i.e. erroneous comments lead to bugs in maintenance.

Interface and Integration Bugs

External interface bugs include invalid timing or sequence assumptions related to external signals, misunderstanding external input and output formats, and user interface bugs. Internal interface bugs include input and output format bugs, inadequate protection against corrupted data, wrong subroutine call sequence, call parameter bugs, and misunderstood entry or exit parameter values. Integration bugs result from inconsistencies or incompatibilities between modules discussed in the form of interface bugs. There may be bugs in data transfer and data sharing between the modules.

System Bugs

There may be bugs while testing the system as a whole based on various parameters like performance, stress, compatibility, usability, etc. For example, in a real-time system, stress testing is very important, as the system must work under maximum load. If the system is put under maximum load at every factor like maximum number of users, maximum memory limit, etc. and if it fails, then there are system bugs.

Testing Bugs

One can question the presence of bugs in the testing phase because this phase is dedicated to finding bugs. But the fact is that bugs are present in testing phase also. After all, testing is also performed by testers – humans. Some testing mistakes are: failure to notice/report a problem, failure to use the most promising test case, failure to make it clear how to reproduce the problem, failure to check for unresolved problems just before the release, failure to verify fixes, failure to provide summary report.

2.1.8 Testing Principles

Now it is time to learn the testing principles that are largely based on the discussion covered in the first chapter and the present one. These principles can be seen as guidelines for a tester.

Effective testing, not exhaustive testing All possible combinations of tests become so large that it is impractical to test them all. So considering the domain of testing as infinite, exhaustive testing is not possible. Therefore, the tester's approach should be based on effective testing to adequately cover program logic and all conditions in the component level design.

Testing is not a single phase performed in SDLC Testing is not just an activity performed after the coding in SDLC. As discussed, the testing phase after coding is just a part of the whole testing process. Testing process starts as soon

as the specifications for the system are prepared and it continues till the release of the product.

Destructive approach for constructive testing Testers must have the psychology that bugs are always present in the program and they must think about the technique of how to uncover them (this is their art of creativity). This psychology of being always suspicious about bugs is a negative/destructive approach. However, it has been proved that such a destructive approach helps in performing constructive and effective testing. Thus, the criterion to have a successful testing is to discover more and more bugs, and not to show that the system does not contain any bugs.

Early testing is the best policy When is the right time to start the testing process? As discussed earlier and we will explore later that testing process is not a phase after coding, rather it starts as soon as requirement specifications are prepared. Moreover, the cost of bugs can be reduced tenfold, as bugs are harder to detect in later stages if they go undetected. Thus, the policy in testing is to start as early as possible.

Probability of existence of an error in a section of a program is proportional to the number of errors already found in that section Suppose the history of a software is that you found 50 errors in Module X, 12 in Module Y, and 3 in Module Z. The software was debugged but after a span of time, we find some errors again and the software is given to a tester for testing. Where should the tester concentrate to find the bugs? This principle says that the tester should start with Module X which has the history of maximum errors. Another way of stating it is that errors seem to come in clusters. The principle provides us the insight that if some sections are found error-prone in testing, then our next testing effort should give priority to these error-prone sections.

Testing strategy should start at the smallest module level and expand towards the whole program This principle supports the idea of incremental testing. Testing must begin at the unit or module level, gradually progressing towards integrated modules and finally the whole system. Testing cannot be performed directly on the whole system. It must start with the individual modules and slowly integrate the modules and test them. After this, the whole system should be tested.

Testing should also be performed by an independent team When programmers develop the software, they test it at their individual modules. However, these programmers are not good testers of their own software. They are basically constructors of the software, but testing needs a destructive approach.

Programmers always think positively that their code does not contain bugs. Moreover, they are biased towards the correct functioning of the specified requirements and not towards detecting bugs. Therefore, it is always recommended to have the software tested by an independent testing team. Testers associated with the same project can also help in this direction, but this is not effective. For effective testing, the software may also be sent outside the organization for testing.

Everything must be recorded in software testing As mentioned earlier, testing is not an intuitive process; rather it is a planned process. It demands that every detail be recorded and documented. We must have the record of every test case run and the bugs reported. Even the inputs provided during testing and the corresponding outputs are to be recorded. Executing the test cases in a recorded and documented way can greatly help while observing the bugs. Moreover, observations can be a lesson for other projects. So the experience with the test cases in one project can be helpful in other projects.

Invalid inputs and unexpected behaviour have a high probability of finding an error Whenever the software is tested, we test for valid inputs and for the functionality that the software is supposed to do. But thinking in a negative way, we must test the software with invalid inputs and the behaviour which is not expected in general. This is also a part of effective testing.

Testers must participate in specification and design reviews Testers' role is not only to get the software and documents and test them. If they are not participating in other reviews like specification and design, it may be possible that either some specifications are not tested or some test cases are built for no specifications.

Let us consider a program. Let S be the set of specified behaviours of the program, P be the implementation of the program, and T be the set of test cases. Now consider the following cases (see Fig. 2.7):

(i) There may be specified behaviours that are not tested (regions 2 and 5).

(ii) Test cases that correspond to unspecified behaviours (regions 3 and 7).

(iii) Program behaviours that are not tested (regions 2 and 6).

The good view of testing is to enlarge the area of region 1. Ideally, all three sets S, P, and T must overlap each other such that all specifications are implemented and all implemented specifications are tested. This is possible only when the test team members participate in all discussions regarding specifications and design.

Figure 2.7 Venn diagram for S, P, T

2.2 Software Testing Life Cycle (STLC)

Since we have recognized software testing as a process, like SDLC, there is need for a well-defined series of steps to ensure successful and effective software testing. This systematic execution of each step will result in saving time and effort. Moreover, the chances are that more number of bugs will be uncovered.

The testing process divided into a well-defined sequence of steps is termed as *software testing life cycle* (STLC). The major contribution of STLC is to involve the testers at early stages of development. This has a significant benefit in the project schedule and cost. The STLC also helps the management in measuring specific milestones.

STLC consists of the following phases (see Fig. 2.8):

Figure 2.8 Software testing life cycle

Test Planning

The goal of test planning is to take into account the important issues of testing strategy, viz. resources, schedules, responsibilities, risks, and priorities, as a roadmap. Test planning issues are in tune with the overall project planning. Broadly, following are the activities during test planning:

- Defining the test strategy.
- Estimate the number of test cases, their duration, and cost.
- Plan the resources like the manpower to test, tools required, documents required.
- Identifying areas of risks.
- Defining the test completion criteria.
- Identification of methodologies, techniques, and tools for various test cases.
- Identifying reporting procedures, bug classification, databases for testing, bug severity levels, and project metrics.

Based on the planning issues as discussed above, analysis is done for various testing activities. The major output of test planning is the *test plan document*. Test plans are developed for each level of testing. After analysing the issues, the following activities are performed:

- Develop a test case format.
- Develop test case plans according to every phase of SDLC.
- Identify test cases to be automated (if applicable).
- Prioritize the test cases according to their importance and criticality.
- Define areas of stress and performance testing.
- Plan the test cycles required for regression testing.

Test Design

One of the major activities in testing is the design of test cases. However, this activity is not an intuitional process; rather it is a well-planned process.

The test design is an important phase after test planning. It includes the following critical activities.

Determining the test objectives and their prioritization This activity decides the broad categories of things to test. The test objectives reflect the fundamental elements that need to be tested to satisfy an objective. For this purpose, you need to gather reference materials like software requirements specification and design documentation. Then on the basis of reference materials, a team

of experts compile a list of test objectives. This list should also be prioritized depending upon the scope and risk.

Preparing list of items to be tested The objectives thus obtained are now converted into lists of items that are to be tested under an objective.

Mapping items to test cases After making a list of items to be tested, there is a need to identify the test cases. A matrix can be created for this purpose, identifying which test case will be covered by which item. The existing test cases can also be used for this mapping. Thus it permits reusing the test cases. This matrix will help in:

(a) Identifying the major test scenarios.

(b) Identifying and reducing the redundant test cases.

(c) Identifying the absence of a test case for a particular objective and as a result, creating them.

Designing the test cases demands a prior analysis of the program at functional or structural level. Thus, the tester who is designing the test cases must understand the cause-and-effect connections within the system intricacies. But look at the rule quoted by Tsuneo Yamaura–*There is only one rule in designing test cases: Cover all features, but do not make too many test cases.*

Some attributes of a good test case are given below:

(a) A good test case is one that has been designed keeping in view the criticality and high-risk requirements in order to place a greater priority upon, and provide added depth for testing the most important functions [12].

(b) A good test case should be designed such that there is a high probability of finding an error.

(c) Test cases should not overlap or be redundant. Each test case should address a unique functionality, thereby not wasting time and resources.

(c) Although it is sometimes possible to combine a series of tests into one test case, a good test case should be designed with a modular approach so that there is no complexity and it can be reused and recombined to execute various functional paths. It also avoids masking of errors and duplication of test-creation efforts [7, 12].

(d) A successful test case is one that has the highest probability of detecting an as-yet-undiscovered error [2].

Selection of test case design techniques While designing test cases, there are two broad categories, namely *black-box testing* and *white-box testing*. Black-

box test case design techniques generate test cases without knowing the internal working of a system. These will be discussed later in this chapter. The techniques to design test cases are selected such that there is more coverage and the system detects more bugs.

Creating test cases and test data The next step is to create test cases based on the testing objectives identified. The test cases mention the objective under which a test case is being designed, the inputs required, and the expected outputs. While giving input specifications, test data must also be chosen and specified with care, as this may lead to incorrect execution of test cases.

Setting up the test environment and supporting tools The test created above needs some environment settings and tools, if applicable. So details like hardware configurations, testers, interfaces, operating systems, and manuals must be specified during this phase.

Creating test procedure specification This is a description of how the test case will be run. It is in the form of sequenced steps. This procedure is actually used by the tester at the time of execution of test cases.

Thus, the hierarchy for test design phase includes: developing test objectives, identifying test cases and creating their specifications, and then developing test case procedure specifications as shown in Fig. 2.9. All the details specified in the test design phase are documented in the test design specification. This document provides the details of the input specifications, output specifications, environmental needs, and other procedural requirements for the test case.

Figure 2.9 Test case design steps

Test Execution

In this phase, all test cases are executed including verification and validation. Verification test cases are started at the end of each phase of SDLC. Validation test cases are started after the completion of a module. It is the decision of the test team to opt for automation or manual execution. Test results are documented in the test incident reports, test logs, testing status, and test summary reports, as shown in Fig. 2.10. These will be discussed in detail in Chapter 9.

Figure 2.10 Documents in test execution

Responsibilities at various levels for execution of the test cases are outlined in Table 2.1.

Table 2.1 Testing level vs responsibility

Test Execution Level	Person Responsible
Unit	Developer of the module
Integration	Testers and Developers
System	Testers, Developers, End-users
Acceptance	Testers, End-users

Post-Execution/Test Review

As we know, after successful test execution, bugs will be reported to the concerned developers. This phase is to analyse bug-related issues and get feedback so that maximum number of bugs can be removed. This is the primary goal of all test activities done earlier.

As soon as the developer gets the bug report, he performs the following activities:

Understanding the bug The developer analyses the bug reported and builds an understanding of its whereabouts.

Reproducing the bug Next, he confirms the bug by reproducing the bug and the failure that exists. This is necessary to cross-check failures. However, some bugs are not reproducible which increases the problems of developers.

Analysing the nature and cause of the bug After examining the failures of the bug, the developer starts debugging its symptoms and tracks back to the actual location of the error in the design. The process of debugging has been discussed in detail in Chapter 17.

After fixing the bug, the developer reports to the testing team and the modified portion of the software is tested once again.

After this, the results from manual and automated testing can be collected. The final bug report and associated metrics are reviewed and analysed for overall testing process. The following activities can be done:

- *Reliability analysis* can be performed to establish whether the software meets the predefined reliability goals or not. If so, the product can be released and the decision on a release date can be taken. If not, then the time and resources required to reach the reliability goals are outlined.
- *Coverage analysis* can be used as an alternative criterion to stop testing.
- *Overall defect analysis* can identify risk areas and help focus our efforts on quality improvement.

2.3 Software Testing Methodology

Software testing methodology is the organization of software testing by means of which the test strategy and test tactics are achieved, as shown in Fig. 2.11. All the terms related to software testing methodology and a complete testing strategy is discussed in this section.

Figure 2.11 Testing methodology

2.3.1 Software Testing Strategy

Testing strategy is the planning of the whole testing process into a well-planned series of steps. In other words, strategy provides a roadmap that includes very specific activities that must be performed by the test team in order to achieve a specific goal.

In Chapter 1, risk reduction was described as one of the goals of testing. In fact, when a test strategy is being developed, risk reduction is addressed first. The components of a testing strategy are discussed below:

Test Factors

Test factors are risk factors or issues related to the system under development. Risk factors need to be selected and ranked according to a specific system under development. The testing process should reduce these test factors to a prescribed level.

Test Phase

This is another component on which the testing strategy is based. It refers to the phases of SDLC where testing will be performed. Testing strategy may be different for different models of SDLC, e.g. strategies will be different for waterfall and spiral models.

2.3.2 Test Strategy Matrix

A test strategy matrix identifies the concerns that will become the focus of test planning and execution. In this way, this matrix becomes an input to develop the testing strategy. The matrix is prepared using test factors and test phase (Table 2.2). The steps to prepare this matrix are discussed below.

Select and rank test factors Based on the test factors list, the most appropriate factors according to specific systems are selected and ranked from the most significant to the least. These are the rows of the matrix.

Identify system development phases Different phases according to the adopted development model are listed as columns of the matrix. These are called test phases.

Identify risks associated with the system under development In the horizontal column under each of the test phases, the test concern with the strategy used to address this concern is entered. The purpose is to identify the concerns that need to be addressed under a test phase. The risks may include any events, actions, or circumstances that may prevent the test program from being implemented or executed according to a schedule, such as late budget

approvals, delayed arrival of test equipment, or late availability of the software application. As risks are identified, they must be assessed for impact and then mitigated with strategies for overcoming them, because risks may be realized despite all precautions having been taken. The test team must carefully examine risks in order to derive effective test and mitigation strategies for a particular application. Concerns should be expressed as questions so that the test strategy becomes a high-level focus for testers when they reach the phase where it's most appropriate to address a concern.

Table 2.2 Test strategy matrix

Test Factors	Test Phase					
	Requirements	Design	Code	Unit test	Integration test	System test

Example 2.1

Creating a test strategy Let's take a project as an example. Suppose a new operating system has to be designed, which needs a test strategy. The following steps are used (Table 2.3).

Table 2.3 Example test strategy matrix

Test Factors	Test Phase					
	Requirements	Design	Code	Unit test	Integration test	System test
Portability	Is portability feature mentioned in specifications according to different hardware?					Is system testing performed on MIPS and INTEL platforms?
Service Level	Is time frame for booting mentioned?	Is time frame incorporated in design of the module?				

Select and rank test factors A critical factor to be considered for the development of an operating system is portability. This is the effort required to transfer a program from one hardware configuration to another. This factor matters the most, as the operating system has to be compatible with most hardware configurations.

Identify the test phases In this step, all test phases affected by the selected test factors are identified. All the affected test phases can be seen according to the test factors in Table 2.3.

Identify the risks associated with each test factor and its corresponding test phase All the risks are basic concerns associated with each factor in a phase, and are expressed in the form of a question. For example, 'Is testing performed successfully on INTEL, MIPS H/W platforms?'

Plan the test strategy for every risk identified After identifying the risks, it is required to plan a strategy to tackle them. It helps testers to start working on testing so that risks are mitigated.

Test strategies generally must concentrate on risks associated with cost overruns, schedule slippage, critical software errors, and other failures. Thus test-strategy design is developed with risks, constrained resources, time limits, and budget restrictions.

2.3.3 Development of Test Strategy

When the project under consideration starts and progresses, testing too starts from the first step of SDLC. Therefore, the test strategy should be such that the testing process continues till the implementation of project. Moreover, the rule for development of a test strategy is that testing 'begins from the smallest unit and progresses to enlarge'. This means the testing strategy should start at the component level and finish at the integration of the entire system. Thus, a test strategy includes testing the components being built for the system, and slowly shifts towards testing the whole system. This gives rise to two basic terms—*Verification* and *Validation*—the basis for any type of testing. It can also be said that the testing process is a combination of verification and validation.

The purpose of verification is to check the software with its specification at every development phase such that any defect can be detected at an early stage of testing and will not be allowed to propagate further. That is why verification can be applied to all stages of SDLC. So *verification* refers to the set of activities that ensures correct implementation of functions in a software. However, as we progress down to the completion of one module or system development, the scope of verification decreases. The validation process starts replacing the verification in the later stages of SDLC. *Validation* is a very general term to test the software as a whole in conformance with customer expectations. According to Boehm [5]–

Verification is 'Are we building the product right?'

Validation is 'Are we building the right product?'

You can relate verification and validation to every complex task of daily life. You divide a complex task into many sub-tasks. In this case, every sub-task is developed and accomplished towards achieving the complex task. Here, you check every sub-task to ensure that you are working in the right direction. This is *verification*. After the sub-tasks have been completed and merged, the entire task is checked to ensure the required task goals have been achieved. This is *validation*.

Verification is checking the work at intermediate stages to confirm that the project is moving in the right direction, towards the set goal.

When a module is prepared with various stages of SDLC like plan, design and code, it is verified at every stage. But there may be various modules in the project. These need to be integrated, after which the full system is built. However, if we simply integrate the modules and build the system for the customer, then we are leaving open spaces for the bugs to intrude. Therefore, after building individual modules, the following stages need to be tested: the module as a whole, integration of modules, and the system built after integration. This is *validation testing*.

2.3.4 Testing Life Cycle Model

Verification and validation (V&V) are the building blocks of a testing process. The formation of test strategy is based on these two terms only. V&V can be best understood when these are modeled in the testing process. This model is known as the Testing Life Cycle Model. Life cycle involves continuous testing of the system during the development process. At predetermined points, the results of the development process are inspected to determine the correctness of implementation. These inspections identify defects as early as possible. But, life cycle testing is dependent upon the completion of predetermined deliverables at a specified point in the development life cycle. It also shows the paths for various types of testing.

V-Testing Life Cycle Model

In V-testing concept [4], as the development team attempts to implement the software, the testing team concurrently starts checking the software. When the project starts, both the system development and the system test process begin. The team that is developing the system begins the system development process and the team that is conducting the system test begins planning the system test process, as shown in Fig. 2.12. Both teams start at the same point using the same information. If there are risks, the tester develops a process to minimize or eliminate them.

Figure 2.12 V-testing model

Figure 2.12 shown above can also be expanded. In Fig. 2.13, on the left arm of the V, the development cycle is progressing and on the right arm, the corresponding testing stages are moving. As said earlier, in the early stages of SDLC, testing comprises more verification activities and towards the later stages, the emphasis is on validation.

Figure 2.13 Expanded V-testing model

The V&V process, in a nutshell, involves (i) verification of every step of SDLC (all verification activities are explained in the next chapter) and (ii) validation of the verified system at the end.

2.3.5 Validation Activities

Validation has the following three activities which are also known as the *three levels of validation testing.*

Unit Testing

It is a major validation effort performed on the smallest module of the system. If avoided, many bugs become latent bugs and are released to the customer. Unit testing is a basic level of testing which cannot be overlooked, and confirms the behaviour of a single module according to its functional specifications.

Integration Testing

It is a validation technique which combines all unit-tested modules and performs a test on their aggregation. One may ask, when we have tested all modules in unit testing, where is the need to test them on aggregation? The answer is *interfacing.* Unit modules are not independent, and are related to each other by interface specifications between them. When we unit test a module, its interfacing with other modules remain untested. When one module is combined with another in an integrated environment, interfacing between units must be tested. If some data structures, messages, or other things are common between some modules, then the standard format of these interfaces must be checked during integration testing, otherwise these will not be able to interface with each other.

But how do we integrate the units together? Is it a random process? It is actually a systematic technique for combining modules. In fact, interfacing among modules is represented by the system design. We integrate the units according to the design and availability of units. Therefore, the tester must be aware of the system design.

System Testing

This testing level focuses on testing the entire integrated system. It incorporates many types of testing, as the full system can have various users in different environments. The purpose is to test the validity for specific users and environments. The validity of the whole system is checked against the requirement specifications.

2.3.6 Testing Tactics

The ways to perform various types of testing under a specific test strategy are discussed below.

Software Testing Techniques

In the previous sections, it has been observed that complete or exhaustive testing is not possible. Instead, our effort should be on effective testing. How-

ever, effective testing is a real challenge in the domain of testing. At this stage, testing can be defined as the design of effective test cases such that most of the testing domains will be covered detecting the maximum number of bugs. As Myers [2] said–

> *Given constraints on time, cost, computer time, etc., the key issue of testing becomes – What subset of all possible test cases has the highest probability of detecting the most errors?*

Therefore, the next objective is to find the technique which will meet both the objectives of effective test case design, i.e. coverage of testing domain and detection of maximum number of bugs. The technique used to design effective test case is called *Software Testing Technique.*

Till now, we have discussed the overall strategy for testing methodology. V&V and the levels of testing under this process describe only the organization and planning of software testing. Actual methods for designing test cases, i.e. software testing techniques, implement the test cases on the software. These techniques can be categorized into two parts: (a) static testing and (b) dynamic testing.

Static Testing

It is a technique for assessing the structural characteristics of source code, design specifications or any notational representation that conforms to well-defined syntactic rules [16]. It is called as *static* because we never execute the code in this technique. For example, the structure of code is examined by the teams but the code is not executed.

Dynamic Testing

All the methods that execute the code to test a software are known as dynamic testing techniques. In this technique, the code is run on a number of inputs provided by the user and the corresponding results are checked. This type of testing is further divided into two parts: (a) black-box testing and (b) white-box testing.

Black-box testing This technique takes care of the inputs given to a system and the output is received after processing in the system. What is being processed in the system? How does the system perform these operations? Black-box testing is not concerned with these questions. It checks the functionality of the system only. That is why the term black-box is used. It is also known as *functional testing.* It is used for system testing under validation.

White-box testing This technique complements black-box testing. Here, the system is not a black box. Every design feature and its corresponding code is checked logically with every possible path execution. So, it takes care of the structural paths instead of just outputs. It is also known as *structural testing* and is used for unit testing under verification.

Testing Tools

Testing tools provide the option to automate the selected testing technique with the help of tools. A tool is a resource for performing a test process. The combination of tools and testing techniques enables the test process to be performed. The tester should first understand the testing techniques and then go for the tools that can be used with each of the techniques.

2.3.7 Considerations in Developing Testing Methodologies

The considerations in developing a testing methodology are described below [4].

Determine Project Risks

A test strategy is developed with the help of another team familiar with the business risks associated with the software. The major issue in determining the strategy is to identify the risks associated with the project. What are the high priority risks? What are the consequences, if risks are not handled?

Determine the Type of Development Project

The environment or methodology to develop the software also affects the testing risks. The risks associated with a new development project will be different from a maintenance project or a purchased software.

Identify Test Activities According to SDLC Phase

After identifying all the risk factors in an SDLC phase, testing can be started in that phase. However, all the testing activities must be recognized at all the SDLC phases.

Build the Test Plan

A tactical test plan is required to start testing. This test plan provides:

- Environment and pre-test background
- Overall objectives
- Test team
- Schedule and budget showing detailed dates for the testing
- Resource requirements including equipment, software, and personnel
- Testing materials including system documentation, software to be tested, test inputs, test documentation, test tools
- Specified business functional requirements and structural functions to be tested
- Type of testing technique to be adopted in a particular SDLC phase or what are the specific tests to be conducted in a particular phase

Since the internal design of a software may be composed of many components, each of these components or units must also have its own test plan. The idea is that units should not be submitted without testing. So an extra effort must be put in preparing unit test plans also.

SUMMARY

Software testing fundamentals have been discussed in this chapter. There is a lot of confusion about using the testing terms. Some terms are used interchangeably, while they imply different meanings. This chapter makes all the related definitions clear. Since the basis for software testing are bugs, their complete life cycle, right from initiation to post-execution, has been described.

The testing principles have been elaborated which will guide the reader in performing every activity related to testing and help in understanding the subsequent chapters. Software testing has matured as a process, thus testing life cycle and its models have been described. Finally, a testing methodology that consists of a roadmap for performing tests has been provided. It will help the reader understand the hierarchy in the test strategy and what is to be done at every phase of testing.

Let us review the important concepts described in this chapter:

- Failure of a program is the mismatch between expected and actual results.
- Fault is a condition in the program that causes failures. This is also known as bug or defect.
- Error is the actual location in the program where a mistake has been made that produced the bug.
- Test case is a well-documented procedure designed to test the functionality of a feature in the system.
- The documents created during testing activities are known as Testware.
- An incident is the symptom(s) associated with a failure that alerts the user to the occurrence of a failure.
- An oracle is the means to judge the success or failure of a test.
- A bug propagates to all the stages of SDLC, if not detected.
- An undetected bug is harder and costlier to detect and debug at a later stage. The earlier you detect, the better.
- Bugs can be classified based on their criticality as: (a) critical, (b) major, (c) medium, and (d) minor.
- Bugs can be classified based on their SDLC phase in which they have been introduced as: (a) requirement and specification bugs, (b) design bugs, (c) coding bugs, (d) interface & integration bugs, (e) system bugs, and (f) testing bugs.
- Software testing life cycle consists of the following phases: (a) test planning, (b) test design, (c) test execution, and (d) post-implementation/test review.

- Software testing methodology is the organization of software testing by means of which the test strategy and test tactics are achieved.
- Testing strategy is planning the whole testing process into a well-planned series of steps.
- A test strategy matrix identifies the concerns that will become the focus of test planning and execution. The matrix is prepared using the test factors and test phase.
- The test strategy has two parts: Verification and Validation
- Verification is to check the software with its specification at every development phase such that any defect can be detected at an early stage of testing.
- Validation is to test the software as a whole in conformance with customer expectations.
- V-testing model emphasizes the concept of early testing. In this model, the scope of verification is more in the early phases and decreases slowly as one module is ready. After this, the scope of validation increases.
- There are three validation activities, generally known as levels of testing: Unit testing, Integration testing, and System testing.
- There are two types of testing techniques: Static and Dynamic testing.
- Static testing is to test the software and its documents without executing the software.
- Dynamic testing is to test the software by executing it.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Fault is synonymous with the word _____.
   (a) Failure
   (b) Defect
   (c) Error
   (d) All of the above
2. The inability of a system or component to perform a required function according to its specification is called as _____.
   (a) Failure
   (b) Bug
   (c) Error
   (d) None of the above
3. Testware includes _____.
   (a) test planning document
   (b) test data
   (c) test specifications
   (d) All of the above

4. Symptom(s) associated with a failure that alerts the user to the occurrence of a failure is called _____.
   (a) Bug
   (b) Error
   (c) Defect
   (d) Incident
5. Testing process starts as soon as the _____ for the system are prepared.
   (a) Design
   (b) Coding
   (c) Specifications
   (d) None of the above
6. Testing strategy should start at the _____ module level and expand towards the whole program.
   (a) Smallest
   (b) Largest
   (c) None of the above
7. Testing is a _____ process.
   (a) Intuitive
   (b) Random
   (c) Planned
   (d) None of the above
8. Planning the whole testing process into a well-planned series of steps is called _____.
   (a) Test strategy matrix
   (b) Test factor
   (c) Test phase
   (d) Test strategy
9. The test strategy matrix is prepared using the _____.
   (a) test planning and execution
   (b) test factor and test phase
   (c) test factor
   (d) test phase
10. The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase is called _____.
   (a) Verification
   (b) Validation
   (c) SDLC
   (d) None of the above

11. The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies the specified requirements is called _____.
    (a) Verification
    (b) Validation
    (c) SDLC
    (d) None of the above
12. In the early stages of SDLC, testing comprises more _____ activities and towards the later stages, the emphasis is on the _____ activities.
    (a) verification, validation
    (b) validation, verification
    (c) integration, coding
    (d) None
13. Technique for assessing the structural characteristics of source code, design specifications, or any notational representation that conforms to well-defined syntactic rules is called _____.
    (a) Dynamic testing
    (b) Static Testing
    (c) Black-Box Testing
    (d) None of the above
14. Every design feature and its corresponding code is checked logically with every possible path execution in _____.
    (a) Black-box testing
    (b) White-box testing
    (c) Testing tool
    (d) None of the above

Review Questions

1. Differentiate error, bug, defect, fault, failure, testware, and incident, giving examples of each.
2. How many states does a bug have?
3. Take a live project and demonstrate the fact that 'Bugs in earlier stages go undetected and propagate'.
4. Collect examples of some commercial products that prove the statement 'Bug affects the economics of Software Testing'
5. Give examples of each category of bug classified based on criticality.
6. Give examples of each category of bug classified based on SDLC.
7. How do you support destructive approach for software testing?

8. What are the benefits of early testing?
9. Demonstrate the fact that in a live project—'The probability of existence of an error in a section of a program is proportional to the number of errors already found in that section.'
10. What will happen if we test the whole system directly?
11. What is the psychology behind testing by an independent team?
12. 'Invalid inputs and unexpected behaviour have a high probability of finding an error.' Prove it in a live project.
13. What is the significance of enlarging the area of region 1 in Fig. 2.7?
14. Explain the different stages of STLC?
15. What is the use of test strategy matrix? Explain with an example other than provided in this chapter.
16. What is the difference between
    (a) Verification and Validation
    (b) Static and Dynamic testing
    (c) Black-box and White-box testing

    Explain with examples.
17. Take a project and identify the Test Activities according to SDLC phases.
18. What are the features of V-testing model? Explain in detail.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

3

Verification and Validation

OBJECTIVES

After reading this chapter, you should be able to understand:

- V-diagram provides the basis for every type of software testing
- Various verification activities
- Various validation activities
- How to perform verification at each stage of SDLC
- Various validation test plans

In the last chapter, it has been seen that a testing strategy may be developed in the form of verification and validation, as shown in the V-diagram (see Fig. 3.1).

A V-diagram provides the following insights about software testing:

- Testing can be implemented in the same flow as for the SDLC.
- Testing can be broadly planned in two activities, namely verification and validation.
- Testing must be performed at every step of SDLC.
- V-diagram supports the concept of early testing.
- V-diagram supports parallelism in the activities of developers and testers.

Figure 3.1 V-Testing

- The more you concentrate on the V&V process, more will be the cost-effectiveness of the software.
- Testers should be involved in the development process.

Since testing is a critical element of quality, verification and validation also relate to the quality of the software. So it can be said that all testing activities can be seen in the form of verification and validation.

In this chapter, the V-diagram has been expanded to get a clear vision of verification and validation activities in SDLC. This expanded V-diagram called the V&V activities diagram gives us some more insights.

3.1 Verification and Validation (V&V) Activities

V&V activities can be understood in the form of a diagram which is described here. To understand this diagram, we first need to understand SDLC phases. After this, verification and validation activities in those SDLC phases will be described. The following SDLC phases [2] (see Fig. 3.2) are considered:

Figure 3.2 SDLC Phases

Requirements gathering The needs of the user are gathered and translated into a written set of requirements. These requirements are prepared from the user's viewpoint only and do not include any technicalities according to the developer.

Requirement specification or objectives In this phase, all the user requirements are specified in developer's terminology. The specified objectives from the full system which is going to be developed, are prepared in the form of a document known as *software requirement specification* (SRS).

Functional design or high-level design Functional design is the process of translating user requirements into a set of external interfaces. The output of the process is the functional design specification, which describes the product's behaviour as seen by an observer external to the product. The *high-level design* is prepared with SRS and software analysts convert the requirements into a usable product. In HLD, the software system architecture is prepared and broken into independent modules. Thus, an HLD document will contain the following items at a macro level:

1. Overall architecture diagrams along with technology details
2. Functionalities of the overall system with the set of external interfaces
3. List of modules
4. Brief functionality of each module
5. Interface relationship among modules including dependencies between modules, database tables identified along with key elements

Internal design or low-level design Since HLD provides the macro-level details of a system, an HLD document cannot be given to programmers for coding. So the analysts prepare a micro-level design document called *internal design* or *low-level design* (LLD). This document describes each and every module in an elaborate manner, so that the programmer can directly code the program based on this. There may be at least one separate document for each module.

Coding If an LLD document is prepared for every module, then it is easy to code the module. Thus in this phase, using design document for a module, its coding is done.

After understanding all the SDLC phases, we need to put together all the verification activities. As described earlier, verification activities are performed

almost at every phase, therefore all the phases described above will be verified, as shown in Fig. 3.3. Along with these verification activities performed at every step, the tester needs to prepare some test plans which will be used in validation activities performed after coding the system. These test plans are prepared at every SDLC phase (see Fig. 3.3).

Figure 3.3 Tester performs verification and prepares test plan during every SDLC phase

When the coding is over for a unit or a system, and parallel verification activities have been performed, then the system can be validated. It means validation activities can be performed now. These are executed with the help of test plans prepared by the testers at every phase of SDLC. This makes the complete V&V activities diagram (see Fig. 3.4) which is discussed in detail in the following sections.

Figure 3.4 V&V diagram

3.2 Verification

We have seen that verification is a set of activities that ensures correct implementation of specific functions in a software. What is the need of verification? Can't we test the software in the final phase of SDLC? Under the V&V process, it is mandatory that verification is performed at every step of SDLC. Verification is needed for the following:

- If verification is not performed at early stages, there are always a chance of mismatch between the required product and the delivered product. Suppose, if requirements are not verified, then it leads to situations where requirements and commitments are not clear. These things become important in case of non-functional requirements and they increase the probability of bugs.

- Verification exposes more errors.
- Early verification decreases the cost of fixing bugs.
- Early verification enhances the quality of software.

After understanding the need of verification, the goals of verification can be formalized which are discussed below [17].

Everything Must be Verified

In principle, all the SDLC phases and all the products of these processes must be verified.

Results of Verification May Not be Binary

Verification may not just be the acceptance or rejection of a product. There are many verification activities whose results cannot be reduced to a binary answer. Often, one has to accept approximations. For example, sometimes correctness of a requirement cannot be rejected or accepted outright, but can be accepted with a degree of satisfaction or rejected with a certain degree of modification.

Even Implicit Qualities Must be Verified

The qualities desired in the software are explicitly stated in the SRS. But those requirements which are implicit and not mentioned anywhere must also be verified.

3.2.1 Verification Activities

All the verification activities are performed in connection with the different phases of SDLC. The following verification activities have been identified:

- Verification of Requirements and Objectives
- Verification of High-Level Design
- Verification of Low-Level Design
- Verification of Coding (Unit Verification)

3.3 Verification of Requirements

In this type of verification, all the requirements gathered from the user's viewpoint are verified. For this purpose, an acceptance criterion is prepared. An acceptance criterion defines the goals and requirements of the proposed system and acceptance limits for each of the goals and requirements. The acceptance criteria matter the most in case of real-time systems where perfor-

mance is a critical issue in certain events. For example, if a real-time system has to process and take decision for a weapon within *x* seconds, then any performance below this would lead to rejection of the system. Therefore, it can be concluded that acceptance criteria for a requirement must be defined by the designers of the system and should not be overlooked, as they can create problems while testing the system.

The tester works in parallel by performing the following two tasks:

1. The tester reviews the acceptance criteria in terms of its completeness, clarity, and testability. Moreover, the tester understands the proposed system well in advance so that necessary resources can be planned for the project [13].
2. The tester prepares the *Acceptance Test Plan* which is referred at the time of *Acceptance Testing* (discussed later).

3.3.1 Verification of Objectives

After gathering requirements, specific objectives are prepared considering every specification. These objectives are prepared in a document called software requirement specification (SRS). In this activity also, two parallel activities are performed by the tester:

1. The tester verifies all the objectives mentioned in SRS. The purpose of this verification is to ensure that the user's needs are properly understood before proceeding with the project.
2. The tester also prepares the *System Test Plan* which is based on SRS. This plan will be referenced at the time of *System Testing* (discussed later).

In verifying the requirements and objectives, the tester must consider both functional and non-functional requirements. Functional requirements may be easy to comprehend. But non-functional requirements pose a challenge to testers in terms of understanding, quantifying, test planning, and test execution.

3.3.2 How to Verify Requirements and Objectives?

Requirement and objectives verification has a high potential of detecting bugs. Therefore, requirements must be verified. As stated above, the testers use the SRS for verification of objectives. One characteristic of a good SRS is that it can be verified. An SRS [18] can be verified, if and only if, every requirement stated herein can be verified. A requirement can be verified, if, and only if, there is some procedure to check that the software meets its requirement. It is a good idea to specify the requirements in a quantification manner. It means that ambiguous statements or language like 'good quality', 'usually', 'may

happen' should be avoided. Instead of this, quantified specifications should be provided. An example of a verifiable statement is

'Module *x* will produce output within 15 sec of its execution.'

OR

'The output should be displayed like this: **TRACK A's speed is *x***'.

It is clear now that verification starts from the requirement phase and every requirement specified in the SRS must be verified. But what are the points against which verification of requirement will be done? Following are the points against which every requirement in SRS should be verified:

Correctness

There are no tools or procedures to measure the correctness of a specification. The tester uses his or her intelligence to verify the correctness of requirements. Following are some points which can be adopted (these points can change according to situation):

(a) Testers should refer to other documentations or applicable standards and compare the specified requirement with them.

(b) Testers can interact with customers or users, if requirements are not well-understood.

(c) Testers should check the correctness in the sense of realistic requirement. If the tester feels that a requirement cannot be realized using existing hardware and software technology, it means that it is unrealistic. In that case, the requirement should either be updated or removed from SRS.

Unambiguous

A requirement should be verified such that it does not provide too many meanings or interpretations. It should not create redundancy in specifications. Each characteristic should be described using a single term, otherwise ambiguity or redundancy may cause bugs in the design phase. The following must be verified:

(a) Every requirement has only one interpretation.

(b) Each characteristic of the final product is described using a single unique term.

Consistent

No specification should contradict or conflict with another. Conflicts produce bugs in the next stages, therefore they must be checked for the following:

(a) Real-world objects conflict, for example, one specification recommends mouse for input, another recommends joystick.

(b) Logical conflict between two specified actions, e.g. one specification requires the function to perform square root, while another specification requires the same function to perform square operation.

(c) Conflicts in terminology should also be verified. For example, at one place, the term *process* is used while at another place, it has been termed as *task* or *module*.

Completeness

The requirements specified in the SRS must be verified for completeness. We must

(a) Verify that all significant requirements such as functionality, performance, design constraints, attribute, or external interfaces are complete.

(b) Check whether responses of every possible input (valid & invalid) to the software have been defined.

(c) Check whether figures and tables have been labeled and referenced completely.

Updation

Requirement specifications are not stable, they may be modified or another requirement may be added later. Therefore, if any updation is there in the SRS, then the updated specifications must be verified.

(a) If the specification is a new one, then all the above mentioned steps and their feasibility should be verified.

(b) If the specification is a change in an already mentioned specification, then we must verify that this change can be implemented in the current design.

Traceability

The traceability of requirements must also be verified such that the origin of each requirement is clear and also whether it facilitates referencing in future development or enhancement documentation. The following two types of traceability must be verified:

Backward traceability Check that each requirement references its source in previous documents.

Forward traceability Check that each requirement has a unique name or reference number in all the documents. Forward traceability assumes more meaning than this, but for the sake of clarity, here it should be understood in the sense that every requirement has been recognized in other documents.

3.4 Verification of High-Level Design

All the requirements mentioned in the SRS document are addressed in this phase and work in the direction of designing the solution. The architecture and design is documented in another document called the *software design document* (SDD).

Like the verification of requirements, the tester is responsible for two parallel activities in this phase as well:

1. The tester verifies the high-level design. Since the system has been decomposed in a number of sub-systems or components, the tester should verify the functionality of these components. Since the system is considered a black box with no low-level details considered here, the stress is also on how the system will interface with the outside world. So all the interfaces and interactions of user/customer (or any person who is interfacing with system) are specified in this phase. The tester verifies that all the components and their interfaces are in tune with requirements of the user. Every requirement in SRS should map the design.
2. The tester also prepares a *Function Test Plan* which is based on the SRS. This plan will be referenced at the time of *Function Testing* (discussed later).

The tester also prepares an *Integration Test Plan* which will be referred at the time of integration testing (discussed later).

3.4.1 How to Verify High-Level Design?

High-level design takes the second place in SDLC, wherein there is a high probability of finding bugs. Therefore, high-level design must be verified as a next step in early testing. Unless the design is specified in a formal way, design cannot be verified. So SDD is referred for design verification. IEEE [19] has provided the standard way of documenting the design in an SDD.

If a bug goes undetected in the high-level design phase, then its cost of fixing increases with every phase. Therefore, verification for high-level design must be done very carefully. This design is divided in three parts (as described by Pressman [7]).

Data Design

It creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). The structure of data has always been an important part of software design. At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to create high-quality applications.

Architectural Design

It focuses on the representation of the structure of software components, their properties, and interactions.

Interface Design

It creates an effective communication medium between the interfaces of different software modules, interfaces between the software system and any other external entity, and interfaces between a user and the software system. Following a set of interface design principles, the design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

Verification of high-level design will consider all the three designs mentioned above and this will become the basis for effective verification.

Verification of Data Design

The points considered for verification of data design are as follows:

- Check whether the sizes of data structure have been estimated appropriately.
- Check the provisions of overflow in a data structure.
- Check the consistency of data formats with the requirements.
- Check whether data usage is consistent with its declaration.
- Check the relationships among data objects in data dictionary.
- Check the consistency of databases and data warehouses with the requirements specified in SRS.

Verification of Architectural Design

The points considered for the verification of architectural design are:

- Check that every functional requirement in the SRS has been take care of in this design.
- Check whether all exceptions handling conditions have been taken care of.
- Verify the process of transform mapping and transaction mapping, used for the transition from requirement model to architectural design.
- Since architectural design deals with the classification of a system into sub-systems or modules, check the functionality of each module according to the requirements specified.
- Check the inter-dependence and interface between the modules.

- In the modular approach of architectural design, there are two issues with modularity–*Module Coupling and Module Cohesion.* A good design will have low coupling and high cohesion. Testers should verify these factors, otherwise they will affect the reliability and maintainability of the system which are non-functional requirements of the system.

Verification of User-Interface Design

The points to be considered for the verification of user-interface design are:

- Check all the interfaces between modules according to the architecture design.
- Check all the interfaces between software and other non-human producer and consumer of information.
- Check all the interfaces between human and computer.
- Check all the above interfaces for their consistency.
- Check the response time for all the interfaces are within required ranges. It is very essential for the projects related to real-time systems where response time is very crucial.
- For a Help Facility, verify the following:
  (i) The representation of Help in its desired manner
  (ii) The user returns to the normal interaction from Help
- For error messages and warnings, verify the following:
  (i) Whether the message clarifies the problem
  (ii) Whether the message provides constructive advice for recovering from the error
- For typed command interaction, check the mapping between every menu option and their corresponding commands.

3.5 Verification of Low-level Design

In this verification, low-level design phase is considered. The abstraction level in this phase is low as compared to high-level design. In LLD, a detailed design of modules and data are prepared such that an operational software is ready. For this, SDD is preferred where all the modules and their interfaces are defined. Every operational detail of each module is prepared. The details of each module or unit is prepared in their separate SRS and SDD.

Testers also perform the following parallel activities in this phase:

1. The tester verifies the LLD. The details and logic of each module is verified such that the high-level and low-level abstractions are consistent.
2. The tester also prepares the *Unit Test Plan* which will be referred at the time of *Unit Testing* (discussed later).

3.5.1 How to Verify Low-Level Design?

This is the last pre-coding phase where internal details of each design entity are described. For verification, the SRS and SDD of individual modules are referred to. Some points to be considered are listed below:

- Verify the SRS of each module.
- Verify the SDD of each module.
- In LLD, data structures, interfaces, and algorithms are represented by design notations; verify the consistency of every item with their design notations.

Organizations can build a two-way traceability matrix between the SRS and design (both HLD and LLD) such that at the time of verification of design, each requirement mentioned in the SRS is verified. In other words, the traceability matrix provides a one-to-one mapping between the SRS and the SDD.

3.6 How to Verify Code?

Coding is the process of converting LLD specifications into a specific language. This is the last phase when we get the operational software with the source code. People have the impression that testing starts only after this phase. However, it has been observed in the last chapter that testing starts as soon as the requirement specifications are given and testers perform parallel activities during every phase of SDLC. If we start testing after coding, then there is a possibility that requirement specifications may have bugs and these might have been passed into the design and consequently into the source code. Therefore, the operational software which is ready now, is not reliable and when bugs appear after this phase, they are very expensive to fix.

Since low-level design is converted into source code using some language, there is a possibility of deviation from the LLD. Therefore, the code must also be verified. The points against which the code must be verified are:

- Check that every design specification in HLD and LLD has been coded using traceability matrix.
- Examine the code against a language specification checklist.

- Code verification can be done most efficiently by the developer, as he has prepared the code. He can verify every statement, control structure, loop, and logic such that every possible method of execution is tested. In this way, he verifies the whole module which he has developed. Some points against which the code can be verified are:
  (a) Misunderstood or incorrect arithmetic precedence
  (b) Mixed mode operations
  (c) Incorrect initialization
  (d) Precision inaccuracy
  (e) Incorrect symbolic representation of an expression
  (f) Different data types
  (g) Improper or non-existent loop termination
  (h) Failure to exit

Two kinds of techniques are used to verify the coding: (a) static testing, and (b) dynamic testing.

Static testing techniques As discussed in the previous chapter, this technique does not involve actual execution. It considers only static analysis of the code or some form of conceptual execution of the code.

Dynamic testing techniques It is complementary to the static testing technique. It executes the code on some test data. The developer is the key person in this process who can verify the code of his module by using the dynamic testing technique.

3.6.1 Unit Verification

Verification of coding cannot be done for the whole system. Moreover, the system is divided into modules. Therefore, verification of coding means the verification of code of modules by their developers. This is also known as *unit verification testing*. Listed below are the points to be considered while performing unit verification [7]:

- Interfaces are verified to ensure that information properly flows in and out of the program unit under test.
- The local data structure is verified to maintain data integrity.
- Boundary conditions are checked to verify that the module is working fine on boundaries also.
- All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- All error handling paths are tested.

Unit verification is largely *white-box oriented*.

3.7 Validation

As described earlier, validation is a set of activities that ensures the software under consideration has been built right and is traceable to customer requirements. Validation testing is performed after the coding is over.

What is the need for validation? When every step of SDLC has been verified, why do we want to test the product at the end? The reasons are:

- To determine whether the product satisfies the users' requirements, as stated in the requirement specification.
- To determine whether the product's actual behaviour matches the desired behaviour, as described in the functional design specification.
- It is not always certain that all the stages till coding are bug-free. The bugs that are still present in the software after the coding phase need to be uncovered.
- Validation testing provides the last chance to discover bugs, otherwise these bugs will move to the final product released to the customer.
- Validation enhances the quality of software.

3.7.1 Validation Activities

The validation activities are divided into *Validation Test Plan* and *Validation Test Execution* which are described as follows:

Validation Test Plan

It starts as soon as the first output of SDLC, i.e. the SRS, is prepared. In every phase, the tester performs two parallel activities—verification at that phase and the corresponding validation test plan. For preparing a validation test plan, testers must follow the points described below.

- Testers must understand the current SDLC phase.
- Testers must study the relevant documents in the corresponding SDLC phase.
- On the basis of the understanding of SDLC phase and related documents, testers must prepare the related test plans which are used at the time of validation testing. Under test plans, they must prepare a sequence of test cases for validation testing.

The following test plans have been recognized which the testers have already prepared with the incremental progress of SDLC phases:

Acceptance test plan This plan is prepared in the requirement phase according to the acceptance criteria prepared from the user feedback. This plan is used at the time of Acceptance Testing.

System test plan This plan is prepared to verify the objectives specified in the SRS. Here, test cases are designed keeping in view how a complete integrated system will work or behave in different conditions. The plan is used at the time of System Testing.

Function test plan This plan is prepared in the HLD phase. In this plan, test cases are designed such that all the interfaces and every type of functionality can be tested. The plan is used at the time of Function Testing.

Integration test plan This plan is prepared to validate the integration of all the modules such that all their interdependencies are checked. It also validates whether the integration is in conformance to the whole system design. This plan is used at the time of Integration Testing.

Unit test plan This plan is prepared in the LLD phase. It consists of a test plan of every module in the system separately. Unit test plan of every unit or module is designed such that every functionality related to individual unit can be tested. This plan is used at the time of Unit Testing.

Validation Test Execution

Validation test execution can be divided in the following testing activities:

Unit validation testing The testing strategy is to first focus on the smaller building blocks of the full system. One unit or module is the basic building block of the whole software that can be tested for all its interfaces and functionality. Thus, unit testing is a process of testing the individual components of a system. A unit or module must be validated before integrating it with other modules. Unit validation is the first validation activity after the coding of one module is over. The motivation for unit validation as compared to the whole system are as follows:

(a) Since the developer has his attention focused on a smaller building block of the software, i.e. unit or module, it is quite natural to test the unit first.

(b) If the whole software is tested at once, then it is very difficult to trace the bug. Thus, debugging becomes easy with unit testing.

(c) In large-scale projects, a number of modules may be there and each module may be composed of tens of thousands of lines of code. In such a case, it is not possible for a single developer to develop all the modules. There is a team of developers working on separate modules. Sometimes,

some modules are sent to other organizations for development. This requires parallelism in software development. If we did not have the concept of module, this type of parallelism would not have existed. Thus, every module can be developed and tested independently.

Integration testing It is the process of combining and testing multiple components or modules together. The individual tested modules, when combined with other modules, are not tested for their interfaces. Therefore, they may contain bugs in integrated environment. Thus, the intention here is to uncover the bugs that are present when unit tested modules are integrated.

Function testing When the integrated system has been tested, all the specified functions and their external interfaces are tested on the software. Every functionality of the system specified in the functions is tested according to its external specifications. An external specification is a precise description of the software behaviour from the viewpoint of the outside world (e.g. user). Thus, function testing is to explore the bugs related to discrepancies between the actual system behaviour and its functional specifications.

The objective of function test is to measure the quality of the functional (business) components of the system. Tests verify that the system behaves correctly from the user/business perspective and functions according to the requirements, models, or any other design paradigm used to specify the application. The function test must determine if each component or business event:

1. performs in accordance to the specifications,
2. responds correctly to all the conditions that may be presented by incoming events/data,
3. moves the data correctly from one business event to the next (including data stores), and
4. is initiated in the order required to meet the business objectives of the system.

System testing It is different from function testing, as it does not test every function. System testing is actually a series of different tests whose primary purpose is to fully exercise a computer-based system [7]. System testing does not aim to test the specified function, but its intention is to test the whole system on various grounds where bugs may occur. For example, if the software fails in some conditions, how does it recover? Another example is protection from improper penetration, i.e. how secure the whole system is.

Acceptance testing As explained earlier, in the requirement phase of SDLC, acceptance criteria for the system to be developed are mentioned in one contract by the customer. When the system is ready, it can be tested against

the acceptance criteria contracted with the customer. This is called acceptance testing. Thus, acceptance testing can be defined as the process of comparing the final system with the needs of customer as agreed on the acceptance criteria.

Installation testing Once the testing team has given the green signal for producing the software, the software is placed into an operational status where it is installed. The installation testing does not test the system, but it tests the process of making the software system operational.

The installation process must identify the steps required to install the system. For example, it may be required that some files may have to be converted into another format. Or it may be possible that some operating software may be required. Thus, installation testing tests the interface to operating software, related software, and any operating procedures.

All these validation testing activities will be discussed in detail in Chapter 7.

SUMMARY

This chapter illustrates the verification and validation activities through the expanded V-diagram. As a part of the strategy of early testing, verification is an important activity for a software tester. This is why verification activity is performed at every stage of SDLC. It reduces the chances of propagation of bugs to the later stages of SDLC.

At the same time, a software tester plans for validation testing while performing verification. These test plans are used when we start validation after a module coding is over. Validation testing activities are performed till the software is installed in an operational status. In this way, both activities assume their importance as a part of early testing as well as a fully tested software product. Thus, verification and validation become the basis for any type of software testing.

Let us review the important concepts described in this chapter:

- Verification is performed at every stage of SDLC to uncover more and more bugs during the earlier stages
- Verification activities are: Verification of requirements and objectives, Verification of high-level design, Verification of low-level design, Verification of coding (Unit Verification)
- After the completion of every SDLC phase, the tester prepares a test plan corresponding to a particular validation testing. This means, the tester does two tasks in parallel: i) Verification of every stage ii) Planning for Validation
- The test plans and corresponding validation testing are shown below:

SDLC Phase	Test Plan	Validation Testing
Requirement Gathering	Acceptance Test Plan	Acceptance Testing
Requirement Specification/ Objectives	System Test Plan	System Testing
Functional Design/ High-Level Design	Function and Integration Test Plan	Function and Integration Testing
Internal Design/Low-level Design	Unit Test Plan	Unit Testing

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Which of the following is true?
   (a) Testing is performed after coding.
   (b) Testing is performed after the integration of modules.
   (c) Testing starts as soon as the SRS is prepared.
   (d) None of the above
2. V&V diagram includes
   (a) Verification only
   (b) Validation only
   (c) Both verification and validation
   (d) None of the above
3. Which test plan is made corresponding to requirement gathering?
   (a) Acceptance test plan
   (b) System test plan
   (c) Function & Integration test plan
   (d) Unit test plan
4. Which test plan is made corresponding to requirement specifications?
   (a) Acceptance test plan
   (b) System test plan
   (c) Function & Integration test plan
   (d) Unit test plan
5. Which test plan is made corresponding to HLD?
   (a) Acceptance test plan
   (b) System test plan
   (c) Function & Integration test plan
   (d) Unit test plan
6. Which test plan is made corresponding to LLD?
   (a) Acceptance test plan
   (b) System test plan
   (c) Function & Integration test plan
   (d) Unit test plan
7. In the V model of testing, the scope of verification from top to bottom
   (a) Increases
   (b) Decreases
   (c) Remains the same
   (d) None of the above

8. For the verification of requirements, you must check for
   (a) Correctness
   (b) Ambiguity
   (c) Completeness
   (d) All of the above
9. For the verification of high-level design, you must check
   (a) Data design
   (b) Architectural design
   (c) User interface design
   (d) All of the above
10. For verifying architectural design, which is true?
   (a) Check that every functional requirement in the SRS has been take care of in this design.
   (b) Check whether all exceptions handling conditions have been taken care of.
   (c) Check the inter-dependence and interface between the modules.
   (d) All of the above
11. For verifying data design, you must check
   (a) Sizes of data structure
   (b) Size of module
   (c) Overflow in a data structure
   (d) All of the above
12. What is the relation between static and dynamic testing technique?
   (a) Mutually exclusive
   (b) Complementary
   (c) Independent
   (d) None
13. What is the proper sequence of various testing?
   (a) Function testing, integration testing, system testing, acceptance testing, unit testing
   (b) Unit testing, integration testing, function testing, system testing, acceptance testing
   (c) Unit testing, integration testing, system testing, function testing, acceptance testing
   (d) None

Review Questions

1. What are the activities performed by a tester at the time of development of a project?
2. What is a V-diagram? What are its benefits?
3. What is the need for verification?
4. What is the need for validation?
5. What is the role of test plans in a V&V diagram?

6. What are the adverse effects on a project, if verification is not performed?
7. Discuss the goals of verification in a project.
8. In a project, only the SRS and SDD are prepared. Can you start testing at this stage?
9. How would you verify the ambiguous requirements in a project and in which stage of SDLC?
10. What do you mean by backward and forward traceability?
11. Develop a checklist of the common coding errors for the language in which you develop your programs.
12. Develop a table describing each phase and corresponding test plan documents related to the project on which you are working.
13. Evaluate the benefits of verification (early testing) in a project.
14. Evaluate the benefits of validation in a project.
15. Suppose you are a member of development team of a Database maintenance project. How would you verify transform mapping and transaction mapping in this project?
16. What is the difference between unit verification and unit validation?
17. What is the difference between function testing and system testing?
18. What is the difference between system testing and acceptance testing?
19. 'V&V diagram is basis for every type of testing?' Comment on this statement.
20. Take a project and perform verification and validation activities as described in this chapter.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Part 2 Testing Techniques

CHAPTERS

Chapter 4:
Dynamic Testing: Black-Box Testing Techniques

Chapter 5:
Dynamic Testing: White-Box Testing Techniques

Chapter 6:
Static Testing

Chapter 7:
Validation Activities

Chapter 8:
Regression Testing

In Part I, we have discussed the fundamentals of effective software testing and how to develop a testing methodology. After devising a testing strategy, we need various testing techniques so that we can design test cases in order to perform testing. There are two views of testing techniques: one view is to categorize based on verification and validation; another is based on static and dynamic testing. We have seen the general strategy of performing verification and validation in Part I. The techniques for verification and validation have been discussed in this part.

Static testing largely maps to verification and dynamic testing to validation. Static testing is performed without executing the code, and dynamic testing is performed with the execution of code. Dynamic testing techniques, namely black-box and white-box techniques, are very popular. We have tried to include every possible method under these categories.

Regression testing is a big problem in software testing. Whenever we make some modifications, we need to execute all the test cases designed earlier as well as some new cases to check the modifications and whether these changes have affected other parts of the software. It becomes a problem as the test suite becomes too large to test. Regression testing is a hot topic for researchers. We have included this testing in techniques so as to define it properly and seek some techniques to deal with it.

This part will make ground for the following concepts:

- Dynamic testing techniques
- Static testing techniques
- Validation testing techniques
- Regression testing and techniques

The following table lists the techniques grouped under their respective categories.

Testing Category	Techniques
Dynamic testing: Black-Box	Boundary value analysis, Equivalence class partitioning, State table-based testing, Decision table-based testing, Cause-effect graphing technique, Error guessing.
Dynamic testing: White-Box	Basis path testing, Graph matrices, Loop testing, Data flow testing, Mutation testing.
Static testing	Inspection, Walkthrough, Reviews.
Validation testing	Unit testing, Integration testing, Function testing, System testing, Acceptance testing, Installation testing.
Regression testing	Selective retest technique, Test prioritization.

Chapter

4

Dynamic Testing: Black-Box Testing Techniques

Black-box technique is one of the major techniques in dynamic testing for designing effective test cases. This technique considers only the functional requirements of the software or module. In other words, the structure or logic of the software is not considered. Therefore, this is also known as *functional testing*. The software system is considered as a black box, taking no notice of its internal structure, so it is also called as *black-box* testing technique.

It is obvious that in black-box technique, test cases are designed based on functional specifications. Input test data is given to the system, which is a black box to the tester, and results are checked against expected outputs after executing the software, as shown in Fig. 4.1.

OBJECTIVES

After reading this chapter, you should be able to understand:

- Black-box testing ignores the structural details of the software
- Test case designing using black-box techniques
- Boundary value analysis method
- Equivalence class testing method
- State table-based testing method
- Decision table-based testing method
- Cause-effect graphing method
- Error guessing

Figure 4.1 Black-box testing

Black-box testing attempts to find errors in the following categories:

- To test the modules independently
- To test the functional validity of the software so that incorrect or missing functions can be recognized
- To look for interface errors
- To test the system behaviour and check its performance
- To test the maximum load or stress on the system
- To test the software such that the user/customer accepts the system within defined acceptable limits

There are various methods to test a software product using black-box techniques. One method chooses the boundary values of the variables, another makes equivalence classes so that only one test case in that class is chosen and executed. Some methods use the state diagrams of the system to form the black-box test cases, while a few other methods use table structure to organize the test cases. It does not mean that one can pick any of these methods for testing the software. Sometimes a combination of methods is employed for rigorous testing. The objective of any test case is to have maximum coverage and capability to discover more and more errors.

4.1 Boundary Value Analysis (BVA)

An effective test case design requires test cases to be designed such that they maximize the probability of finding errors. BVA technique addresses this issue. With the experience of testing team, it has been observed that test cases designed with boundary input values have a high chance to find errors. It means that most of the failures crop up due to boundary values.

BVA is considered a technique that uncovers the bugs at the boundary of input values. Here, boundary means the maximum or minimum value taken by the input domain. For example, if A is an integer between 10 and 255, then boundary checking can be on 10(9,10,11) and on 255(256,255,254). Similarly, B is another integer variable between 10 and 100, then boundary checking can be on 10(9,10,11) and 100(99,100,101), as shown in Fig. 4.2.

Figure 4.2 Boundary value analysis

BVA offers several methods to design test cases as discussed in the following sections.

4.1.1 Boundary Value Checking (BVC)

In this method, the test cases are designed by holding one variable at its extreme value and other variables at their nominal values in the input domain.

The variable at its extreme value can be selected at:

(a) Minimum value (Min)

(b) Value just above the minimum value (Min^+)

(c) Maximum value (Max)

(d) Value just below the maximum value (Max^-)

Let us take the example of two variables, *A* and *B*. If we consider all the above combinations with nominal values, then following test cases (see Fig. 4.3) can be designed:

1. A_{nom}, B_{min}
2. A_{nom}, B_{min+}
3. A_{nom}, B_{max}
4. A_{nom}, B_{max-}
5. A_{min}, B_{nom}
6. A_{min+}, B_{nom}
7. A_{max}, B_{nom}
8. A_{max-}, B_{nom}
9. A_{nom}, B_{nom}

Figure 4.3 Boundary value checking

It can be generalized that for n variables in a module, $4n + 1$ test cases can be designed with boundary value checking method.

4.1.2 Robustness Testing Method

The idea of BVC can be extended such that boundary values are exceeded as:

- A value just greater than the Maximum value (Max^+)
- A value just less than Minimum value (Min^-)

When test cases are designed considering the above points in addition to BVC, it is called *robustness testing.*

Let us take the previous example again. Add the following test cases to the list of 9 test cases designed in BVC:

10. A_{max+}, B_{nom}
11. A_{min-}, B_{nom}
12. A_{nom}, B_{max+}
13. A_{nom}, B_{min-}

It can be generalized that for n input variables in a module, $6n + 1$ test cases can be designed with robustness testing.

4.1.3 Worst-Case Testing Method

We can again extend the concept of BVC by assuming more than one variable on the boundary. It is called *worst-case testing method*.

Again, take the previous example of two variables, A and B. We can add the following test cases to the list of 9 test cases designed in BVC as:

10. A_{min}, B_{min}
11. A_{min+}, B_{min}
12. A_{min}, B_{min+}
13. A_{min+}, B_{min+}
14. A_{max}, B_{min}
15. A_{max-}, B_{min}
16. A_{max}, B_{min+}
17. A_{max-}, B_{min+}
18. A_{min}, B_{max}
19. A_{min+}, B_{max}
20. A_{min}, B_{max-}
21. A_{min+}, B_{max-}
22. A_{max}, B_{max}
23. A_{max-}, B_{max}
24. A_{max}, B_{max-}
25. A_{max-}, B_{max-}

It can be generalized that for n input variables in a module, 5^n test cases can be designed with worst-case testing.

BVA is applicable when the module to be tested is a function of several independent variables. This method becomes important for physical quantities where boundary condition checking is crucial. For example, systems having requirements of minimum and maximum temperature, pressure or speed, etc. However, it is not useful for Boolean variables.

Example 4.1

A program reads an integer number within the range [1,100] and determines whether it is a prime number or not. Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution

(a) **Test cases using BVC** Since there is one variable, the total number of test cases will be $4n + 1 = 5$.

In our example, the set of minimum and maximum values is shown below:

Min value = 1
Min^+ value = 2
Max value = 100
Max^- value = 99
Nominal value = 50–55

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	53	Prime number

(b) **Test cases using robust testing** Since there is one variable, the total number of test cases will be $6n + 1 = 7$. The set of boundary values is shown below:

Min value = 1
Min^- value = 0
Min^+ value = 2
Max value = 100
Max^- value = 99
Max^+ value = 101
Nominal value = 50–55

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	0	Invalid input
2	1	Not a prime number
3	2	Prime number
4	100	Not a prime number
5	99	Not a prime number
6	101	Invalid input
7	53	Prime number

(c) **Test cases using worst-case testing** Since there is one variable, the total number of test cases will be $5^n = 5$. Therefore, the number of test cases will be same as BVC.

Example 4.2

A program computes a^b where a lies in the range [1,10] and b within [1,5]. Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution

(a) **Test cases using BVC** Since there are two variables, a and b, the total number of test cases will be $4n + 1 = 9$. The set of boundary values is shown below:

	a	b
Min value	1	1
Min$^+$ value	2	2
Max value	10	5
Max$^-$ value	9	4
Nominal value	5	3

Using these values, test cases can be designed as shown below:

Test Case ID	a	b	Expected Output
1	1	3	1
2	2	3	8
3	10	3	1000
4	9	3	729
5	5	1	5
6	5	2	25
7	5	4	625
8	5	5	3125
9	5	3	125

(b) **Test cases using robust testing** Since there are two variables, a and b, the total number of test cases will be $6n + 1 = 13$. The set of boundary values is shown below:

	a	b
Min value	1	1
Min^- value	0	0
Min^+ value	2	2
Max value	10	5
Max^+ value	11	6
Max^- value	9	4
Nominal value	5	3

Using these values, test cases can be designed as shown below:

Test Case ID	a	b	Expected output
1	0	3	Invalid input
2	1	3	1
3	2	3	8
4	10	3	1000
5	11	3	Invalid input
6	9	3	729
7	5	0	Invalid input
8	5	1	5
9	5	2	25
10	5	4	625
11	5	5	3125
12	5	6	Invalid input
13	5	3	125

(c) **Test cases using worst-case testing** Since there are two variables, a and b, the total number of test cases will be $5^n = 25$.

The set of boundary values is shown below:

	a	b
Min value	1	1
Min^+ value	2	2
Max value	10	5
Max^- value	9	4
Nominal value	5	3

There may be more than one variable at extreme values in this case. Therefore, test cases can be designed as shown below:

Test Case ID	a	b	Expected Output
1	1	1	1
2	1	2	1
3	1	3	3
4	1	4	1
5	1	5	1
6	2	1	2
7	2	2	4
8	2	3	8
9	2	4	16
10	2	5	32
11	5	1	5
12	5	2	25
13	5	3	125
14	5	4	625
15	5	5	3125
16	9	1	9
17	9	2	81
18	9	3	729
19	9	4	6561
20	9	5	59049
21	10	1	10
22	10	2	100
23	10	3	1000
24	10	4	10000
25	10	5	100000

Example 4.3

A program reads three numbers, *A*, *B*, and *C*, within the range [1, 50] and prints the largest number. Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution

(a) **Test cases using BVC** Since there are three variables, *A*, *B*, and *C*, the total number of test cases will be $4n + 1 = 13$. The set of boundary values is shown below:

Min value = 1
Min^+ value = 2
Max value = 50
Max^- value = 49
Nominal value = 25–30

Using these values, test cases can be designed as shown below:

Test Case ID	A	B	C	Expected Output
1	1	25	27	C is largest
2	2	25	28	C is largest
3	49	25	25	B and C are largest
4	50	25	29	A is largest
5	25	1	30	C is largest
6	25	2	26	C is largest
7	25	49	27	B is largest
8	25	50	28	B is largest
9	25	28	1	B is largest
10	25	27	2	B is largest
11	25	26	49	C is largest
12	25	26	50	C is largest
13	25	25	25	Three are equal

(b) **Test cases using robust testing** Since there are three variables, *A*, *B*, and *C*, the total number of test cases will be $6n + 1 = 19$.

The set of boundary values is shown below:

Min value = 1
Min^- value = 0
Min^+ value = 2
Max value = 50
Max^+ value = 51
Max^- value = 49
Nominal value = 25–30

Using these values, test cases can be designed as shown below:

Test Case ID	A	B	C	Expected Output
1	0	25	27	Invalid input
2	1	25	27	C is largest
3	2	25	28	C is largest
4	49	25	25	B and C are largest
5	50	25	29	A is largest
6	51	27	25	Invalid input
7	25	0	26	Invalid input
8	25	1	30	C is largest
9	25	2	26	C is largest
10	25	49	27	B is largest
11	25	50	28	B is largest
12	26	51	25	Invalid input
13	25	25	0	Invalid input
14	25	28	1	B is largest
15	25	27	2	B is largest
16	25	26	49	C is largest
17	25	26	50	C is largest
18	25	29	51	Invalid input
19	25	25	25	Three are equal

(c) **Test cases using worst-case testing** Since there are three variables, A, B, and C, the total number of test cases will be $5^n = 125$.

The set of boundary values is shown below:

Min value = 1
Min^+ value = 2
Max value = 50
Max^- value = 49
Nominal value = 25–30

There may be more than one variable at extreme values in this case. Therefore, test cases can be design as shown below:

Test Case ID	A	B	C	Expected Output
1	1	1	1	All three are equal
2	1	1	2	C is greatest
3	1	1	25	C is greatest
4	1	1	49	C is greatest
5	1	1	50	C is greatest
6	1	2	1	B is greatest
7	1	2	2	B and C
8	1	2	25	C is greatest
9	1	2	49	C is greatest
10	1	2	50	C is greatest
11	1	25	1	B is greatest
12	1	27	2	B is greatest
13	1	26	25	B is greatest
14	1	25	49	B is greatest
15	1	27	50	C is greatest
16	1	49	1	B is greatest
17	1	49	2	B is greatest
18	1	49	25	B is greatest
19	1	49	49	B and C
20	1	49	50	C is greatest
21	1	50	1	B is greatest
22	1	50	2	B is greatest
23	1	50	25	B is greatest
24	1	50	49	B is greatest
25	1	50	50	B and C
26	2	1	1	A is largest
27	2	1	2	A and C
28	2	1	25	C is greatest

29	2	1	49	C is greatest
30	2	1	50	C is greatest
31	2	2	1	A and B
32	2	2	2	All three are equal
33	2	2	25	C is greatest
34	2	2	49	C is greatest
35	2	2	50	C is greatest
36	2	25	1	B is greatest
37	2	27	2	B is greatest
38	2	28	25	B is greatest
39	2	26	49	C is greatest
40	2	28	50	C is greatest
41	2	49	1	B is greatest
42	2	49	2	B is greatest
43	2	49	25	B is greatest
44	2	49	49	B and C
45	2	49	50	C is greatest
46	2	50	1	B is greatest
47	2	50	2	B is greatest
48	2	50	25	B is greatest
49	2	50	49	B is greatest
50	2	50	50	B and C
51	25	1	1	A is greatest
52	25	1	2	A is greatest
53	25	1	25	A and C
54	25	1	49	C is greatest
55	25	1	50	C is greatest
56	25	2	1	A is greatest
57	25	2	2	A is greatest
58	25	2	25	A and C
59	25	2	49	C is greatest
60	25	2	50	C is greatest
61	25	27	1	B is greatest
62	25	26	2	B is greatest
63	25	25	25	All three are equal
64	25	28	49	C is greatest
65	25	29	50	C is greatest
66	25	49	1	B is greatest
67	25	49	2	B is greatest
68	25	49	25	B is greatest

69	25	49	49	B is greatest
70	25	49	50	C is greatest
71	25	50	1	B is greatest
72	25	50	2	B is greatest
73	25	50	25	B is greatest
74	25	50	49	B is greatest
75	25	50	50	B is greatest
76	49	1	1	A is greatest
77	49	1	2	A is greatest
78	49	1	25	A is greatest
79	49	1	49	A and C
80	49	1	50	C is greatest
81	49	2	1	A is greatest
82	49	2	2	A is greatest
83	49	2	25	A is greatest
84	49	2	49	A and C
85	49	2	50	C is greatest
86	49	25	1	A is greatest
87	49	29	2	A is greatest
88	49	25	25	A is greatest
89	49	27	49	A and C
90	49	28	50	C is greatest
91	49	49	1	A and B
92	49	49	2	A and B
93	49	49	25	A and B
94	49	49	49	All three are equal
95	49	49	50	C is greatest
96	49	50	1	B is greatest
97	49	50	2	B is greatest
98	49	50	25	B is greatest
99	49	50	49	B is greatest
100	49	50	50	B and C
101	50	1	1	A is greatest
102	50	1	2	A is greatest
103	50	1	25	A is greatest
104	50	1	49	A is greatest
105	50	1	50	A and C
106	50	2	1	A is greatest
107	50	2	2	A is greatest
108	50	2	25	A is greatest

109	50	2	49	A is greatest
110	50	2	50	A and C
111	50	26	1	A is greatest
112	50	25	2	A is greatest
113	50	27	25	A is greatest
114	50	29	49	A is greatest
115	50	30	50	A and C
116	50	49	1	A is greatest
117	50	49	2	A is greatest
118	50	49	26	A is greatest
119	50	49	49	A is greatest
120	50	49	50	A and C
121	50	50	1	A and B
122	50	50	2	A and B
123	50	50	26	A and B
124	50	50	49	A and B
125	50	50	50	All three are equal

Example 4.4

A program determines the next date in the calendar. Its input is entered in the form of <ddmmyyyy> with the following range:

$1 \leq mm \leq 12$

$1 \leq dd \leq 31$

$1900 \leq yyyy \leq 2025$

Its output would be the next date or it will display 'invalid date.' Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution

(a) **Test cases using BVC** Since there are three variables, month, day, and year, the total number of test cases will be $4n + 1 = 13$. The set of boundary values is shown below:

	Month	Day	Year
Min value	1	1	1900
Min^+ value	2	2	1901
Max value	12	31	2025
Max^- value	11	30	2024
Nominal value	6	15	1962

Using these values, test cases can be designed as shown below:

Test Case ID	Month	Day	Year	Expected Output
1	1	15	1962	16-1-1962
2	2	15	1962	16-2-1962
3	11	15	1962	16-11-1962
4	12	15	1962	16-12-1962
5	6	1	1962	2-6-1962
6	6	2	1962	3-6-1962
7	6	30	1962	1-7-1962
8	6	31	1962	Invalid input
9	6	15	1900	16-6-1900
10	6	15	1901	16-6-1901
11	6	15	2024	16-6-2024
12	6	15	2025	16-6-2025
13	6	15	1962	16-6-1962

(b) **Test cases using robust testing** The total number of test cases will be $6n + 1 = 19$. The set of boundary values is shown below:

	Month	Day	Year
Min^- value	0	0	1899
Min value	1	1	1900
Min^+ value	2	2	1901
Max value	12	31	2025
Max^- value	11	30	2024
Max^+ value	13	32	2026
Nominal value	6	15	1962

Using these values, test cases can be designed as shown below:

Test Case ID	Month	Day	Year	Expected Output
1	0	15	1962	Invalid date
2	1	15	1962	16-1-1962
3	2	15	1962	16-2-1962
4	11	15	1962	16-11-1962
5	12	15	1962	16-12-1962
6	13	15	1962	Invalid date
7	6	0	1962	Invalid date
8	6	1	1962	2-6-1962
9	6	2	1962	3-6-1962
10	6	30	1962	1-7-1962

11	6	31	1962	Invalid input
12	6	32	1962	Invalid date
13	6	15	1899	Invalid date
14	6	15	1900	16-6-1900
15	6	15	1901	16-6-1901
16	6	15	2024	16-6-2024
17	6	15	2025	16-6-2025
18	6	15	2026	Invalid date
19	6	15	1962	16-6-1962

(c) **Test cases using worst-case testing** The total number of test cases will be $5^n = 125$. The set of boundary values is shown below:

	Month	Day	Year
Min value	1	1	1900
Min^+ value	2	2	1901
Max value	12	31	2025
Max^- value	11	30	2024
Nominal value	6	15	1962

Using these values, test cases can be designed as shown below:

Test Case ID	Month	Day	Year	Expected Output
1	1	1	1900	2-1-1900
2	1	1	1901	2-1-1901
3	1	1	1962	2-1-1962
4	1	1	2024	2-1-2024
5	1	1	2025	2-1-2025
6	1	2	1900	3-1-1900
7	1	2	1901	3-1-1901
8	1	2	1962	3-1-1962
9	1	2	2024	3-1-2024
10	1	2	2025	3-1-2025
11	1	15	1900	16-1-1900
12	1	15	1901	16-1-1901
13	1	15	1962	16-1-1962
14	1	15	2024	16-1-2024
15	1	15	2025	16-1-2025
16	1	30	1900	31-1-1900

17	1	30	1901	31-1-1901
18	1	30	1962	31-1-1962
19	1	30	2024	31-1-2024
20	1	30	2025	31-1-2025
21	1	31	1900	1-2-1900
22	1	31	1901	1-2-1901
23	1	31	1962	1-2-1962
24	1	31	2024	1-2-2024
25	1	31	2025	1-2-2025
26	2	1	1900	2-2-1900
27	2	1	1901	2-2-1901
28	2	1	1962	2-2-1962
29	2	1	2024	2-1-2024
30	2	1	2025	2-2-2025
31	2	2	1900	3-2-1900
32	2	2	1901	3-2-1901
33	2	2	1962	3-2-1962
34	2	2	2024	3-2-2024
35	2	2	2025	3-2-2025
36	2	15	1900	16-2-1900
37	2	15	1901	16-2-1901
38	2	15	1962	16-2-1962
39	2	15	2024	16-2-2024
40	2	15	2025	16-2-2025
41	2	30	1900	Invalid date
42	2	30	1901	Invalid date
43	2	30	1962	Invalid date
44	2	30	2024	Invalid date
45	2	30	2025	Invalid date
46	2	31	1900	Invalid date
47	2	31	1901	Invalid date
48	2	31	1962	Invalid date
49	2	31	2024	Invalid date
50	2	31	2025	Invalid date
51	6	1	1900	2-6-1900
52	6	1	1901	2-6-1901
53	6	1	1962	2-6-1962
54	6	1	2024	2-6-2024

55	6	1	2025	2-6-2025
56	6	2	1900	3-6-1900
57	6	2	1901	3-6-1901
58	6	2	1962	3-6-1962
59	6	2	2024	3-6-2024
60	6	2	2025	3-6-2025
61	6	15	1900	16-6-1900
62	6	15	1901	16-6-1901
63	6	15	1962	16-6-1962
64	6	15	2024	16-6-2024
65	6	15	2025	16-6-2025
66	6	30	1900	1-7-1900
67	6	30	1901	1-7-1901
68	6	30	1962	1-7-1962
69	6	30	2024	1-7-2024
70	6	30	2025	1-7-2025
71	6	31	1900	Invalid date
72	6	31	1901	Invalid date
73	6	31	1962	Invalid date
74	6	31	2024	Invalid date
75	6	31	2025	Invalid date
76	11	1	1900	2-11-1900
77	11	1	1901	2-11-1901
78	11	1	1962	2-11-1962
79	11	1	2024	2-11-2024
80	11	1	2025	2-11-2025
81	11	2	1900	3-11-1900
82	11	2	1901	3-11-1901
83	11	2	1962	3-11-1962
84	11	2	2024	3-11-2024
85	11	2	2025	3-11-2025
86	11	15	1900	16-11-1900
87	11	15	1901	16-11-1901
88	11	15	1962	16-11-1962
89	11	15	2024	16-11-2024
90	11	15	2025	16-11-2025
91	11	30	1900	1-12-1900
92	11	30	1901	1-12-1901

93	11	30	1962	1-12-1962
94	11	30	2024	1-12-2024
95	11	30	2025	1-12-2025
96	11	31	1900	Invalid date
97	11	31	1901	Invalid date
98	11	31	1962	Invalid date
99	11	31	2024	Invalid date
100	11	31	2025	Invalid date
101	12	1	1900	2-12-1900
102	12	1	1901	2-12-1901
103	12	1	1962	2-12-1962
104	12	1	2024	2-12-2024
105	12	1	2025	2-12-2025
106	12	2	1900	3-12-1900
107	12	2	1901	3-12-1901
108	12	2	1962	3-12-1962
109	12	2	2024	3-12-2024
110	12	2	2025	3-12-2025
111	12	15	1900	16-12-1900
112	12	15	1901	16-12-1901
113	12	15	1962	16-12-1962
114	12	15	2024	16-12-2024
115	12	15	2025	16-12-2025
116	12	30	1900	31-12-1900
117	12	30	1901	31-12-1901
118	12	30	1962	31-12-1962
119	12	30	2024	31-12-2024
120	12	30	2025	31-12-2025
121	12	31	1900	1-1-1901
122	12	31	1901	1-1-1902
123	12	31	1962	1-1-1963
124	12	31	2024	1-1-2025
125	12	31	2025	1-1-2026

4.2 Equivalence Class Testing

We know that the input domain for testing is too large to test every input. So we can divide or partition the input domain based on a common feature or

a class of data. Equivalence partitioning is a method for deriving test cases wherein classes of input conditions called *equivalence classes* are identified such that each member of the class causes the same kind of processing and output to occur. Thus, instead of testing every input, only one test case from each partitioned class can be executed. It means only one test case in the equivalence class will be sufficient to find errors. This test case will have a representative value of a class which is equivalent to a test case containing any other value in the same class. If one test case in an equivalence class detects a bug, all other test cases in that class have the same probability of finding bugs. Therefore, instead of taking every value in one domain, only one test case is chosen from one class. In this way, testing covers the whole input domain, thereby reduces the total number of test cases. In fact, it is an attempt to get a good *hit rate* to find maximum errors with the smallest number of test cases.

Equivalence partitioning method for designing test cases has the following goals:

Completeness Without executing all the test cases, we strive to touch the completeness of testing domain.

Non-redundancy When the test cases are executed having inputs from the same class, then there is redundancy in executing the test cases. Time and resources are wasted in executing these redundant test cases, as they explore the same type of bug. Thus, the goal of equivalence partitioning method is to reduce these redundant test cases.

To use equivalence partitioning, one needs to perform two steps:

1. Identify equivalence classes
2. Design test cases

4.2.1 Identification of Equivalent Classes

How do we partition the whole input domain? Different equivalence classes are formed by grouping inputs for which the behaviour pattern of the module is similar. The rationale of forming equivalence classes like this is the assumption that if the specifications require exactly the same behaviour for each element in a class of values, then the program is likely to be constructed such that it either succeeds or fails for each value in that class. For example, the specifications of a module that determines the absolute value for integers specify different behaviour patterns for positive and negative integers. In this case, we will form two classes: one consisting of positive integers and another consisting of negative integers [14].

Two types of classes can always be identified as discussed below:

Valid equivalence classes These classes consider valid inputs to the program.

Invalid equivalence classes One must not be restricted to valid inputs only. We should also consider invalid inputs that will generate error conditions or unexpected behaviour of the program, as shown in Fig. 4.4.

Figure 4.4 Equivalence classes

There are no well-defined rules for identifying equivalence classes, as it is a heuristic process. However, some guidelines are defined for forming equivalence classes:

- If there is no reason to believe that the entire range of an input will be treated in the same manner, then the range should be split into two or more equivalence classes.
- If a program handles each valid input differently, then define one valid equivalence class per valid input.
- Boundary value analysis can help in identifying the classes. For example, for an input condition, say $0 \leq a \leq 100$, one valid equivalent class can be formed from the valid range of a. And with BVA, two invalid classes that cross the minimum and maximum values can be identified, i.e. $a < 0$ and $a > 100$.
- If an input variable can identify more than one category, then for each category, we can make equivalent classes. For example, if the input is a character, then it can be an alphabet, a number, or a special character. So we can make three valid classes for this input and one invalid class.
- If the requirements state that the number of items input by the system at some point must lie within a certain range, specify one valid class where the number of inputs is within the valid range, one invalid class where there are very few inputs, and one invalid class where there are too many inputs. For example, specifications state that a maximum of 4 purchase orders can be registered against a product. The equivalence classes are: the valid equivalence class ($1 \leq$ no. of purchase orders ≤ 4), the invalid class (no. of purchase orders > 4), and the invalid class (no. of purchase orders < 1).

- If an input condition specifies a 'must be' situation (e.g., 'first character of the identifier must be a letter'), identify a valid equivalence class (it is a letter) and an invalid equivalence class (it is not a letter).
- Equivalence classes can be of the output desired in the program. For an output equivalence class, the goal is to generate test cases such that the output for that test case lies in the output equivalence class. Determining test cases for output classes may be more difficult, but output classes have been found to reveal errors that are not revealed by just considering the input classes.
- Look for membership of an input condition in a set or group and identify valid (within the set) and invalid (outside the set) classes. For example, if the requirements state that a valid province code is ON, QU, and NB, then identify: the valid class (code is one of ON, QU, NB) and the invalid class (code is not one of ON, QU, NB).
- If the requirements state that a particular input item match a set of values and each case will be dealt with differently, identify a valid equivalence class for each element and only one invalid class for values outside the set. For example, if a discount code must be input as *P* for a preferred customer, *R* for a standard reduced rate, or *N* for none, and if each case is treated differently, identify: the valid class code = *P*, the valid class code = *R*, the valid class code = *N*, the invalid class code is not one of *P*, *R*, *N*.
- If an element of an equivalence class will be handled differently than the others, divide the equivalence class to create an equivalence class with only these elements and an equivalence class with none of these elements. For example, a bank account balance may be from 0 to Rs 10 lakh and balances of Rs 1,000 or more are not subject to service charges. Identify: the valid class: (0 ≤ balance < Rs 1,000), i.e. balance is between 0 and Rs 1,000 – not including Rs 1,000; the valid class: (Rs 1,000 ≤ balance ≤ Rs 10 lakh, i.e. balance is between Rs 1,000 and Rs 10 lakh inclusive the invalid class: (balance < 0) the invalid class: (balance > Rs 10 lakh).

4.2.2 Identifying the Test Cases

A few guidelines are given below to identify test cases through generated equivalence classes:

- Assign a unique identification number to each equivalence class.
- Write a new test case covering as many of the uncovered valid equivalence classes as possible, until all valid equivalence classes have been covered by test cases.

- Write a test case that covers one, and only one, of the uncovered invalid equivalence classes, until all invalid equivalence classes have been covered by test cases. The reason that invalid cases are covered by individual test cases is that certain erroneous-input checks mask or supersede other erroneous-input checks. For instance, if the specification states 'Enter type of toys (Automatic, Mechanical, Soft toy) and amount (1–10000)', the test case [ABC 0] expresses two error (invalid inputs) conditions (invalid toy type and invalid amount) will not demonstrate the invalid amount test case, hence the program may produce an output 'ABC is unknown toy type' and not bother to examine the remainder of the input.

Remember that there may be many possible solutions for one problem in this technique, depending on the criteria chosen for partitioning the test domain.

Example 4.5

A program reads three numbers, *A*, *B*, and *C*, with a range [1, 50] and prints the largest number. Design test cases for this program using equivalence class testing technique.

Solution

1. First we partition the domain of input as valid input values and invalid values, getting the following classes:

$$I_1 = \{<A,B,C> : 1 \le A \le 50\}$$
$$I_2 = \{<A,B,C> : 1 \le B \le 50\}$$
$$I_3 = \{<A,B,C> : 1 \le C \le 50\}$$
$$I_4 = \{<A,B,C> : A < 1\}$$
$$I_5 = \{<A,B,C> : A > 50\}$$
$$I_6 = \{<A,B,C> : B < 1\}$$
$$I_7 = \{<A,B,C> : B > 50\}$$
$$I_8 = \{<A,B,C> : C < 1\}$$
$$I_9 = \{<A,B,C> : C > 50\}$$

Now the test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class.

The test cases are shown below:

Test case ID	A	B	C	Expected result	Classes covered by the test case
1	13	25	36	C is greatest	I_1, I_2, I_3
2	0	13	45	Invalid input	I_4
3	51	34	17	Invalid input	I_5
4	29	0	18	Invalid input	I_6
5	36	53	32	Invalid input	I_7
6	27	42	0	Invalid input	I_8
7	33	21	51	Invalid input	I_9

2. We can derive another set of equivalence classes based on some possibilities for three integers, A, B, and C. These are given below:

$$I_1 = \{<A,B,C> : A > B,\ A > C\}$$
$$I_2 = \{<A,B,C> : B > A,\ B > C\}$$
$$I_3 = \{<A,B,C> : C > A,\ C > B\}$$
$$I_4 = \{<A,B,C> : A = B,\ A \neq C\}$$
$$I_5 = \{<A,B,C> : B = C,\ A \neq B\}$$
$$I_6 = \{<A,B,C> : A = C,\ C \neq B\}$$
$$I_7 = \{<A,B,C> : A = B = C\}$$

Test case ID	A	B	C	Expected Result	Classes Covered by the test case
1	25	13	13	A is greatest	I_1, I_5
2	25	40	25	B is greatest	I_2, I_6
3	24	24	37	C is greatest	I_3, I_4
4	25	25	25	All three are equal	I_7

Example 4.6

A program determines the next date in the calendar. Its input is entered in the form of <ddmmyyyy> with the following range:

$1 \leq \text{mm} \leq 12$

$1 \leq \text{dd} \leq 31$

$1900 \leq \text{yyyy} \leq 2025$

Its output would be the next date or an error message 'invalid date.' Design test cases using equivalence class partitioning method.

Solution

First we partition the domain of input in terms of valid input values and invalid values, getting the following classes:

$$I_1 = \{<m, d, y> : 1 \le m \le 12\}$$
$$I_2 = \{<m, d, y> : 1 \le d \le 31\}$$
$$I_3 = \{<m, d, y> : 1900 \le y \le 2025\}$$
$$I_4 = \{<m, d, y> : m < 1\}$$
$$I_5 = \{<m, d, y> : m > 12\}$$
$$I_6 = \{<m, d, y> : d < 1\}$$
$$I_7 = \{<m, d, y> : d > 31\}$$
$$I_8 = \{<m, d, y> : y < 1900\}$$
$$I_9 = \{<m, d, y> : y > 2025\}$$

The test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class. The test cases are shown below:

Test case ID	mm	dd	yyyy	Expected result	Classes covered by the test case
1	5	20	1996	21-5-1996	I_1, I_2, I_3
2	0	13	2000	Invalid input	I_4
3	13	13	1950	Invalid input	I_5
4	12	0	2007	Invalid input	I_6
5	6	32	1956	Invalid input	I_7
6	11	15	1899	Invalid input	I_8
7	10	19	2026	Invalid input	I_9

Example 4.7

A program takes an angle as input within the range [0, 360] and determines in which quadrant the angle lies. Design test cases using equivalence class partitioning method.

Solution

1. First we partition the domain of input as valid and invalid values, getting the following classes:

$$I_1 = \{<\text{Angle}> : 0 \le \text{Angle} \le 360\}$$
$$I_2 = \{<A,B,C> : \text{Angle} < 0\}$$
$$I_3 = \{<A,B,C> : \text{Angle} > 0\}$$

The test cases designed from these classes are shown below:

Test Case ID	Angle	Expected results	Classes covered by the test case
1	50	I Quadrant	I_1
2	–1	Invalid input	I_2
3	361	Invalid input	I_3

2. The classes can also be prepared based on the output criteria as shown below:

$$O_1 = \{\text{<Angle>: First Quadrant, if } 0 \le \text{Angle} \le 90\}$$
$$O_2 = \{\text{<Angle>: Second Quadrant, if } 91 \le \text{Angle} \le 180\}$$
$$O_3 = \{\text{<Angle>: Third Quadrant, if } 181 \le \text{Angle} \le 270\}$$
$$O_4 = \{\text{<Angle>: Fourth Quadrant, if } 271 \le \text{Angle} \le 360\}$$
$$O_5 = \{\text{<Angle>: Invalid Angle}\};$$

However, O_5 is not sufficient to cover all invalid conditions this way. Therefore, it must be further divided into equivalence classes as shown below:

$$O_{51} = \{\text{<Angle>: Invalid Angle, if Angle} < 0\}$$
$$O_{52} = \{\text{<Angle>: Invalid Angle, if Angle} > 360\}$$

Now the test cases can be designed from the above derived classes as shown below:

Test Case ID	Angle	Expected results	Classes covered by the test case
1	50	I Quadrant	O_1
2	135	II Quadrant	O_2
3	250	III Quadrant	O_3
4	320	IV Quadrant	O_4
5	370	Invalid angle	O_{51}
6	–1	Invalid angle	O_{52}

4.3 State Table-Based Testing

Tables are useful tools for representing and documenting many types of information relating to test case design. These are beneficial for the applications which can be described using state transition diagrams and state tables. First we define some basic terms related to state tables which are discussed below.

4.3.1 Finite State Machine (FSM)

An FSM is a behavioural model whose outcome depends upon both previous and current inputs. FSM models can be prepared for software structure or software behaviour. And it can be used as a tool for functional testing. Many testers prefer to use FSM model as a guide to design functional tests.

4.3.2 State Transition Diagrams or State Graph

A system or its components may have a number of states depending on its input and time. For example, a task in an operating system can have the following states:

1. **New State**: When a task is newly created.
2. **Ready**: When the task is waiting in the ready queue for its turn.
3. **Running**: When instructions of the task are being executed by CPU.
4. **Waiting**: When the task is waiting for an I/O event or reception of a signal.
5. **Terminated**: The task has finished execution.

States are represented by *nodes*. Now with the help of nodes and transition links between the nodes, a state transition diagram or state graph is prepared. A state graph is the pictorial representation of an FSM. Its purpose is to depict the states that a system or its components can assume. It shows the events or circumstances that cause or result from a change from one state to another [21].

Whatever is being modeled is subjected to inputs. As a result of these inputs, when one state is changed to another, it is called a *transition*. Transitions are represented by links that join the nodes. The state graph of task states is shown in Fig. 4.5.

Figure 4.5 State graph

Each arrow link provides two types of information:

1. Transition events like admitted, dispatch, interrupt, etc.

2. The resulting output from a state like T1, T2, T3, etc.

   T0 = Task is in new state and waiting for admission to ready queue

   T1 = A new task admitted to ready queue

   T2 = A ready task has started running

   T3 = Running task has been interrupted

   T4 = Running task is waiting for I/O or event

   T5 = Wait period of waiting task is over

   T6 = Task has completed execution

4.3.3 State Table

State graphs of larger systems may not be easy to understand. Therefore, state graphs are converted into tabular form for convenience sake, which are known as state tables. State tables also specify states, inputs, transitions, and outputs. The following conventions are used for state table [9]:

- Each row of the table corresponds to a state.
- Each column corresponds to an input condition.
- The box at the intersection of a row and a column specifies the next state (transition) and the output, if any.

The state table for task states is given in Table 4.1.

Table 4.1 State table

State\Input Event	Admit	Dispatch	Interrupt	I/O or Event Wait	I/O or Event Wait Over	Exit
New	**Ready/ T1**	New / T0	New / T0	New / T0	New / T0	New / T0
Ready	Ready/ T1	**Running/ T2**	Ready / T1	Ready / T1	Ready / T1	Ready / T1
Running	Running/T2	Running/ T2	**Ready / T3**	**Waiting/ T4**	Running/ T2	**Terminated/T6**
Waiting	Waiting/T4	Waiting / T4	Waiting/T4	Waiting / T4	**Ready / T5**	Waiting / T4

The highlighted cells of the table are valid inputs causing a change of state. Other cells are invalid inputs which do not cause any transition in the state of a task.

4.3.4 State Table-based Testing

After reviewing the basics, we can start functional testing with state tables. A state graph and its companion state table contain information that is converted into test cases.

The procedure for converting state graphs and state tables into test cases is discussed below.

1. **Identify the states** The number of states in a state graph is the number of states we choose to recognize or model. In practice, the state is directly or indirectly recorded as a combination of values of variables that appear in the database. As an example, the state could be composed of the values of a counter whose possible values ranged from 0 to 9, combined with the setting of two bit flags, leading to a total of $2 \times 2 \times 10 = 40$ states. When the state graph represents an explicit state table implementation, this value is encoded so that bugs in the number of states are less likely; but the encoding can be wrong. Failing to account for all the states is one of the more common bugs in the software that can be modeled by state graphs. As an explicit state table mechanization is not typical, the opportunities for missing states abound. Find the number of states as follows [9]:
   - Identify all the component factors of the state.
   - Identify all the allowable values for each factor.
   - The number of states is the product of the number of allowable values of all the factors.
2. **Prepare state transition diagram after understanding transitions between states** After having all the states, identify the inputs on each state and transitions between states and prepare the state graph. Every input state combination must have a specified transition. If the transition is impossible, then there must be a mechanism that prevents that input from occurring in that state.
   A program cannot have contradictions or ambiguities. Ambiguities are impossible because the program will do something for every input. Even if the state does not change, by definition, this is a transition to the same state. A seeming contradiction could come about in a model if all the factors that constitute the state and all the inputs are not taken care of. If someone as a designer says while debugging, 'sometimes it works and sometimes it doesn't', it means there is a state factor about which he is not aware—a factor probably caused by a bug. Exploring the real state graph and recording the transitions and outputs for each combination of input and state may lead to discovering the bug.
3. **Convert the state graph into the state table as discussed earlier**
4. **Analyse the state table for its completeness**
5. **Create the corresponding test cases from the state table**
   Test cases are produced in a tabular form known as the *test case table* which contains six columns as shown below [22]:

Test case ID : a unique identifier for each test case

Test Source : a trace back to the corresponding cell in the state table

Current State : the initial condition to run the test

Event : the input triggered by the user

Output : the current value returned

Next State : the new state achieved

The test cases derived from the state table (Table 4.1) are shown below in Table 4.2.

Table 4.2 Deriving test cases from state table

Test Case ID	Test Source	Input		Expected Results	
		Current State	Event	Output	Next State
TC1	Cell 1	New	Admit	T1	Ready
TC2	Cell 2	New	Dispatch	T0	New
TC3	Cell 3	New	Interrupt	T0	New
TC4	Cell 4	New	I/O wait	T0	New
TC5	Cell 5	New	I/O wait over	T0	New
TC6	Cell 6	New	exit	T0	New
TC7	Cell 7	Ready	Admit	T1	Ready
TC8	Cell 8	Ready	Dispatch	T2	Running
TC9	Cell 9	Ready	Interrupt	T1	Ready
TC10	Cell 10	Ready	I/O wait	T1	Ready
TC11	Cell 11	Ready	I/O wait	T1	Ready
TC12	Cell 12	Ready	Exit	T1	Ready
TC13	Cell 13	Running	Admit	T2	Running
TC14	Cell 14	Running	Dispatch	T2	Running
TC15	Cell 15	Running	Interrupt	T3	Ready
TC16	Cell 16	Running	I/O wait	T4	Waiting
TC17	Cell 17	Running	I/O wait over	T2	Running
TC18	Cell 18	Running	Exit	T6	Terminated
TC19	Cell 19	Waiting	Admit	T4	Waiting
TC20	Cell 20	Waiting	Dispatch	T4	Waiting
TC21	Cell 21	Waiting	Interrupt	T4	Waiting
TC22	Cell 22	Waiting	I/O wait	T4	Waiting
TC23	Cell 23	Waiting	I/O wait over	T5	Ready
TC24	Cell 24	Waiting	Exit	T4	Waiting

Test case TC1 gets information from Cell 1 and indicates that one task is created *New* in the system. This new task is getting the input event *Admit* and transitions to the new state *Ready* and the output is T1 (i.e. task is present in Ready queue). Similarly other test cases can be derived. From the state graph, we can recognize all input sequences and can form some detailed test cases also.

4.4 Decision Table-Based Testing

Boundary value analysis and equivalence class partitioning methods do not consider combinations of input conditions. These consider each input separately. There may be some critical behaviour to be tested when some combinations of input conditions are considered.

Decision table is another useful method to represent the information in a tabular method. It has the specialty to consider complex combinations of input conditions and resulting actions. Decision tables obtain their power from logical expressions. Each operand or variable in a logical expression takes on the value, TRUE or FALSE.

4.4.1 Formation of Decision Table

A decision table is formed with the following components (see Table 4.3):

Table 4.3 Decision table structure

ENTRY

		Rule 1	Rule 2	Rule 3	Rule 4	...
Condition Stub	C1	True	True	False	I	
	C2	False	True	False	True	
	C3	True	True	True	I	
Action Stub	A1		X			
	A2	X			X	
	A3			X		

Condition stub It is a list of input conditions for which the complex combination is made.

Action stub It is a list of resulting actions which will be performed if a combination of input condition is satisfied.

Condition entry It is a specific entry in the table corresponding to input conditions mentioned in the condition stub. When we enter TRUE or FALSE

for all input conditions for a particular combination, then it is called a *Rule*. Thus, a rule defines which combination of conditions produces the resulting action. When the condition entry takes only two values—TRUE or FALSE, then it is called *Limited Entry Decision Table*. When the condition entry takes several values, then it is called *Extended Entry Decision Table*. In limited entry decision table, condition entry, which has no effect whether it is True or False, is called a *Don't-Care* state or *immaterial state* (represented by *1*). The state of a don't-care condition does not affect the resulting action.

Action entry It is the entry in the table for the resulting action to be performed when one rule (which is a combination of input condition) is satisfied. 'X' denotes the action entry in the table.

The guidelines to develop a decision table for a problem are discussed below [7]:

- List all actions that can be associated with a specific procedure (or module).
- List all conditions (or decision made) during execution of the procedure.
- Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
- Define rules by indicating what action occurs for a set of conditions.

4.4.2 Test Case Design using Decision Table

For designing test cases from a decision table, following interpretations should be done:

- Interpret condition stubs as the inputs for the test case.
- Interpret action stubs as the expected output for the test case.
- Rule, which is the combination of input conditions, becomes the test case itself.
- If there are k rules over n binary conditions, there are at least k test cases and at the most 2^n test cases.

Example 4.8

A program calculates the total salary of an employee with the conditions that if the working hours are less than or equal to 48, then give normal salary. The hours over 48 on normal working days are calculated at the rate of 1.25 of the salary. However, on holidays or Sundays, the hours are calculated at the rate of 2.00 times of the salary. Design test cases using decision table testing.

Solution

The decision table for the program is shown below:

ENTRY

		Rule 1	Rule 2	Rule3
Condition Stub	C1: Working hours > 48	I	F	T
	C2: Holidays or Sundays	T	F	F
Action Stub	A1: Normal salary		X	
	A2: 1.25 of salary			X
	A3: 2.00 of salary	X		

The test cases derived from the decision table are given below:

Test Case ID	Working Hour	Day	Expected Result
1	48	Monday	Normal Salary
2	50	Tuesday	1.25 of salary
3	52	Sunday	2.00 of salary

Example 4.9

A wholesaler has three commodities to sell and has three types of customers. Discount is given as per the following procedure:

(i) For DGS & D orders, 10% discount is given irrespective of the value of the order.

(ii) For orders of more than Rs 50,000, agents get a discount of 15% and the retailer gets a discount of 10%.

(iii) For orders of Rs 20,000 or more and up to Rs 50,000, agents get 12% and the retailer gets 8% discount.

(iv) For orders of less than Rs 20,000, agents get 8% and the retailer gets 5% discount.

The above rules do not apply to the furniture items wherein a flat rate of 10% discount is admissible to all customers irrespective of the value of the order.

Design test cases for this system using decision table testing.

Solution

		ENTRY							
		R1	R2	R3	R4	R5	R6	R7	R8
Condition Stub	C1: DGS & D	T	F	F	F	F	F	F	F
	C2: Agent	F	T	F	T	F	T	F	I
	C3: Retailer	F	F	T	F	T	F	T	I
	C4: Order > 50,000	I	T	T	F	F	F	F	I
	C5: Order ≥ 20000 to < 50,000	I	F	F	T	T	F	F	I
	C6: Order < 20,000	I	F	F	F	F	T	T	I
	C7: Furniture	F	F	F	F	F	F	F	T
Action Stub	A1: Discount of 5%							X	
	A2: Discount of 8%					X	X		
	A3: Discount of 10%	X		X					X
	A4: Discount of 12%				X				
	A5: Discount of 15%		X						

The test cases derived from the decision table are given below:

Test Case ID	Type of Customer	Product Furniture?	Order Value (Rs)	Expected Result
1	DGS & D	No	51,000	10% Discount
2	Agent	No	52,000	15% Discount
3	Retailer	No	53,000	10% Discount
4	Agent	No	23,000	12% Discount
5	Retailer	No	27,000	8% Discount
6	Agent	No	15,000	8% Discount
7	Retailer	No	18,000	5% Discount
8	Agent	Yes	34,000	10% Discount

Example 4.10

A university is admitting students in a professional course subject to the following conditions:

(a) Marks in Java ≥ 70

(b) Marks in C++ ≥ 60

(c) Marks in OOAD ≥ 60

(d) Total in all three subjects ≥ 220 OR Total in Java and C++ ≥ 150

If the aggregate mark of an eligible candidate is more than 240, he will be eligible for scholarship course, otherwise he will be eligible for normal course. The program reads the marks in the three subjects and generates the following outputs:

(i) Not eligible

(ii) Eligible for scholarship course

(iii) Eligible for normal course

Design test cases for this program using decision table testing.

Solution

ENTRY

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
C1: marks in Java ≥ 70	T	T	T	T	F	I	I	I	T	T
C2: marks in C++ ≥ 60	T	T	T	T	I	F	I	I	T	T
C3: marks in OOAD ≥ 60	T	T	T	T	I	I	F	I	T	T
C4:Total in three subjects ≥ 220	T	F	T	T	I	I	I	F	T	T
C5:Total in Java & C++ ≥ 150	F	T	F	T	I	I	I	F	T	T
C6:Aggregate marks > 240	F	F	T	T	I	I	I	I	F	T
A1: Eligible for normal course	X	X							X	
A2: Eligible for scholarship course			X	X						X
A3: Not eligible					X	X	X	X		

Test Case ID	Java	C++	OOAD	Aggregate Marks	Expected Output
1	70	75	60	224	Eligible for normal course
2	75	75	70	220	Eligible for normal course
3	75	74	91	242	Eligible for scholarship course
4	76	77	89	242	Eligible for scholarship course
5	68	78	80	226	Not eligible
6	78	45	78	201	Not eligible
7	80	80	50	210	Not eligible
8	70	72	70	212	Not eligible
9	75	75	70	220	Eligible for normal course
10	76	80	85	241	Eligible for scholarship course

4.4.3 Expanding the Immaterial Cases in Decision Table

Immaterial cases (I) have been shown in the decision table which are *don't-care conditions*. These conditions mean that the value of a particular condition in the specific rule does not make a difference whether it is TRUE or FALSE. However, we should always test both the values of a don't-care condition. So the rules in the table can be expanded. Sometimes expanding the decision table to spell out don't-care conditions can reveal hidden problems. Original decision table shown till now in the examples above and the expanded table are logically equivalent, implying that the combinations of conditions result in tests that exercise the same circumstances.

Example 4.11

Consider Example 4.8 once again whose decision table is shown below with immaterial cases.

ENTRY

		Rule 1	Rule 2	Rule 3
Condition Stub	C1: Working hours > 48	I	F	T
	C2: Holidays or Sundays	T	F	F
Action Stub	A1: Normal salary		X	
	A2: 1.25 of salary			X
	A3: 2.00 of salary	X		

The immaterial test case in Rule 1 of the above table can be expanded by taking both T and F values of C1. The expanded decision table is shown below:

ENTRY

		Rule 1-1	Rule 1-2	Rule 2	Rule 3
Condition Stub	C1: Working hours > 48	F	T	F	T
	C2: Holidays or Sundays	T	T	F	F
Action Stub	A1: Normal salary			X	
	A2: 1.25 of salary				X
	A3: 2.00 of salary	X	X		

The test cases derived from the expanded decision table are given below:

Test Case ID	Working Hour	Day	Expected Result
1	48	Monday	Normal salary
2	50	Tuesday	1.25 of salary
3	52	Sunday	2.00 of salary
4	30	Sunday	2.00 of salary

4.5 Cause-Effect Graphing Based Testing

As said earlier, boundary value analysis and equivalence class partitioning methods do not consider combinations of input conditions. Like decision tables, cause-effect graphing is another technique for combinations of input conditions. But cause-effect graphing takes the help of decision table to design a test case. Therefore, cause-effect graphing is the technique to represent the situations of combinations of input conditions and then we convert the cause-effect graph into decision table for the test cases.

One way to consider all valid combinations of input conditions is to consider all valid combinations of the equivalence classes of input conditions. This simple approach will result in an unusually large number of test cases, many of which will not be useful for revealing any new errors. For example, if there are n different input conditions, such that a combination is valid, we will have 2^n test cases. Cause-effect graphing techniques help in selecting combinations of input conditions in a systematic way, such that the number of test cases does not become unmanageably large [14].

The following process is used to derive the test cases [2].

Division of specification The specification is divided into workable pieces, as cause-effect graphing becomes complex when used on large specifications.

Identification of causes and effects The next step is to identify *causes and effects* in the specifications. A cause is a distinct input condition identified in the problem. It may also be an equivalence class of input conditions. Similarly, an effect is an output condition.

Transformation of specification into a cause-effect graph Based on the analysis of the specification, it is transformed into a Boolean graph linking the causes and effects. This is the cause-effect graph. Complete the graph by adding the constraints, if any, between causes and effects.

Conversion into decision table The cause-effect graph obtained is converted into a limited-entry decision table by verifying state conditions in the graph. Each column in the table represents a test case.

Deriving test cases The columns in the decision table are converted into test cases.

4.5.1 Basic Notations for Cause-Effect Graph

Identity

According to the identity function, if x is 1, y is 1; else y is 0.

NOT

This function states that if x is 1, y is 0; else y is 1.

OR

The OR function states that if x or y or z is 1, A is 1; else A is 0.

AND

This function states that if both x and y are 1, z is 1; else z is 0.

Exclusive

Sometimes, the specification contains an impossible combination of causes such that two causes cannot be set to 1 simultaneously. For this, Exclusive function is used. According to this function, it always holds that either x or y can be 1, i.e. x and y cannot be 1 simultaneously.

Inclusive

It states that at least one of x, y, and z must always be 1 (x, y, and z cannot be 0 simultaneously).

One and Only One

It states that one and only one of x and y must be 1.

Requires

It states that for x to be 1, y must be 1, i.e. it is impossible for x to be 1 and y to be 0.

Mask

It states that if x is 1, y is forced to 0.

Example 4.12

A program has been designed to determine the nature of roots of a quadratic equation. The quadratic equation takes three input values from the range [0, 100]. Design the test cases using cause-effect graphing technique.

Solution

First we identify the causes and effects in this problem as given below:

C1: $a \neq 0$
C2: $b = 0$
C3: $c = 0$
C4: $D > 0$ where D is $b^2 - 4ac$
C5: $D < 0$
C6: $D = 0$
C7: $a = b = c$
C8: $a = c = b/2$
E1: Not a quadratic equation
E2: Real roots
E3: Imaginary roots
E4: Equal roots

Based on these causes and effects and analysing the constraints, the cause-effect graph is prepared, as shown in Fig. 4.6.

Figure 4.6 Cause-effect graph for Example 4.12

Now this graph is converted into a limited entry decision table as shown below.

	R1	R2	R3	R4	R5	R6	R7
C1: a ≠ 0	T	T	T	T	T	T	F
C2: b = 0	F	I	I	T	F	F	I
C3: c = 0	I	F	I	T	F	F	I
C4: D > 0	T	F	F	F	F	F	I
C5: D < 0	F	T	F	F	T	F	I
C6: D = 0	F	F	T	T	F	T	I
C7: a = b = c	F	I	F	F	T	F	I
C8: a = c = b/2	F	F	I	F	F	T	I
A1: Not a quadratic equation							X
A2: Real roots	X						
A3: Imaginary roots		X			X		
A4: Equal roots			X	X		X	

The test cases designed based on this table are given below.

Test Case ID	a	b	c	Expected Output
1	1	50	50	Real roots
2	100	50	50	Imaginary roots
3	1	6	9	Equal
4	100	0	0	Equal
5	99	99	99	Imaginary
6	50	100	50	Equal
7	0	50	30	Not a quadratic equation

4.6 Error Guessing

Error guessing is the preferred method used when all other methods fail. Sometimes it is used to test some special cases. According to this method, errors or bugs can be guessed which do not fit in any of the earlier defined situations. So test cases are generated for these special cases.

It is a very practical case wherein the tester uses his intuition and makes a guess about where the bug can be. The tester does not have to use any particular testing technique. However, this capability comes with years of experience in a particular field of testing. This is the reason that experienced managers can easily smell out errors as compared to a novice tester.

As discussed earlier, every testing activity is recorded. The history of bugs can help in identifying some special cases in the project. There is a high probability that errors made in a previous project is repeated again. In these situations, error guessing is an *ad hoc approach*, based on intuition, experience, knowledge of project, and bug history. Any of these can help to expose the errors. The basic idea is to make a list of possible errors in error-prone situations and then develop the test cases. Thus, there is no general procedure for this technique, as it is largely an intuitive and ad hoc process.

For example, consider the system for calculating the roots of a quadratic equation. Some special cases in this system are as follows:

- What will happen when $a = 0$? Though, we do consider this case, there are chances that we overlook it while testing, as it has two cases:
  - (i) If $a = 0$ then the equation is no longer quadratic.
  - (ii) For calculation of roots, division is by zero.
- What will happen when all the inputs are negative?
- What will happen when the input list is empty?

SUMMARY

Black-box testing is a dynamic testing technique, wherein the software is tested using a set of inputs and expected outputs. It does not consider the logics of the program and how it has been developed. It considers only the functionality of the program and how it behaves under test. This chapter discusses black-box testing techniques. All the methods have been discussed taking sufficient number of examples such that a new method can be learnt easily and the reader can design the test cases using a particular method.

Let us review the important concepts described in this chapter:

- Black-box testing is a technique, wherein the structure of program is overlooked. The internal logic of the program is unknown to the tester.
- Boundary value analysis is the method to uncover the bugs by looking at the boundary of the inputs used in the program. The idea is that the programmer tests the program by having nominal values of the inputs but does not check its boundary values and thus leaves the bugs.
- Equivalence class partitioning is a method for deriving test cases, wherein classes of input conditions called equivalence classes are identified such that each member of the class causes the same kind of processing and output to occur. It reduces the input domain space and thereby the testing effort.
- State table-based testing is a convenient method for testing systems where states and transitions are specified.
- Decision table is another useful method to represent the information in a tabular method. It has the speciality to consider complex combinations of input conditions and resulting actions.

- Decision table consists of: condition stub, action stub, condition entry, and action entry.
- In a decision table, when we enter TRUE or FALSE for all input conditions for a particular combination, then it is called a Rule.
- In a decision table, when the condition entry takes only two values – TRUE or FALSE, then it is called a Limited Entry Decision Table. When the condition entry takes several values, then it is called an Extended Entry Decision Table.
- Cause-effect graphing is the technique to represent combinations of input conditions.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Black-box testing is a _____.
   (a) Static testing
   (b) Dynamic testing
   (c) None of the above
2. It has been observed that test cases, which are designed with boundary input values, have a _____ chance of finding errors.
   (a) High
   (b) Low
   (c) Medium
   (d) Zero
3. How many test cases are there in BVC if there are 5 variables in a module?
   (a) 23
   (b) 13
   (c) 10
   (d) 21
4. How many test cases are there in robustness testing if there are 5 variables in a module?
   (a) 23
   (b) 31
   (c) 10
   (d) 21
5. How many test cases are there in worst-case testing if there are 4 variables in a module?
   (a) 623
   (b) 513
   (c) 625
   (d) 521

6. Each row of state table corresponds to _____.
   (a) Input
   (b) State
   (c) Transition
   (d) None of the above
7. Each column of state table corresponds to _____.
   (a) Input
   (b) State
   (c) Transition
   (d) None of the above
8. Intersection of a row and a column specifies _____.
   (a) Input
   (b) State
   (c) Transition and output
   (d) None of the above
9. What are the components of a decision table?
   (a) Condition stub
   (b) Condition entry
   (c) Action stub
   (d) All
10. If there are *k* rules over *n* binary conditions, there are at least _____ test cases and at the most _____ test cases.
    (a) $k+2$, 2^{n+2}
    (b) $k+3$, 2^{n+3}
    (c) k, 2^{n}
    (d) None of the above
11. Boundary value analysis and equivalence class partitioning methods do not consider _____.
    (a) Combinations of input conditions
    (b) Inputs
    (c) Outputs
    (d) None

Review Questions

1. What are the types of errors detected by black-box testing?
2. Which type of testing is possible with BVA?
3. Which type of testing is possible with equivalence class partitioning?

4. A program calculates the GCD of three numbers in the range [1, 50]. Design test cases for this program using BVC, robust testing, and worst-case testing methods.
5. A program takes as input a string (5–20 characters) and a single character and checks whether that single character is present in the string or not. Design test cases for this program using BVC, robust testing, and worst-case testing methods.
6. A program reads the data of employees in a company by taking the following inputs and prints them:

   Name of Employee (Max. 15 valid characters A–Z, a–z, space)

   Employee ID (10 characters)

   Designation (up to 20 characters)

   Design test cases for this program using BVC, robust testing, and worst-case testing methods.
7. A mobile phone service provider uses a program that computes the monthly bill of customers as follows:

   Minimum Rs 300 for up to 120 calls

   Plus Re 1 per call for the next 70 calls

   Plus Rs 0.80 per call for the next 50 calls

   Plus Rs 0.40 per call for any call beyond 220 calls.

   Design test cases for this program using equivalence class testing technique.
8. A program reads players' records with the following detail and prints a team-wise list containing player name with their batting average:

   Player name (max. 30 characters)

   Team name (max. 20 characters)

   Batting average

   Design test cases for this program using BVC, robust testing, and worst-case testing methods.
9. Consider Example 4.10 and design test cases for this program using equivalence class testing technique.
10. Consider a system having an FSM for a stack having the following states and transitions:

    States

    Initial: Before creation

    Empty: Number of elements = 0

    Holding: Number of elements > 0, but less than the maximum capacity

    Full: Number elements = maximum

    Final: After destruction

Transitions

Initial to Empty: Create

Empty to Holding, Empty to Full, Holding to Holding, Holding to Full: Add

Empty to Final, Full to Final, Holding to Final: Destroy

Holding to Empty, Full to Holding, Full to Empty: Delete

Design test cases for this FSM using state table-based testing.

11. Passengers who travel more than 50,000 km. per calendar year and in addition, pay cash for tickets or have been traveling regularly for more than eight years are to receive a free round trip ticket around India. Passengers who travel less than 50,000 km. per calendar year and have been availing railway services regularly for more than eight years also get a free round ticket around India.

    Design test cases for this system using decision table testing.

12. Consider Problem 6 and design test cases using cause-effect graphing technique.
13. How do you expand immaterial test cases in decision table testing? Expand the immaterial test cases in Problem 11.
14. 'Error guessing is an ad hoc approach for testing.' Comment on this statement.
15. A program takes as input three angles and determines the type of triangle. If all the three angles are less than 90, it is an acute angled triangle. If one angle is greater than 90, it is an obtuse angled triangle. If one angle is equal to 90, it is a right angled triangle.

    Design test cases for this program using equivalence class testing technique.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

5

Dynamic Testing: White-Box Testing Techniques

White-box testing is another effective testing technique in dynamic testing. It is also known as *glass-box* testing, as everything that is required to implement the software is visible. The entire design, structure, and code of the software have to be studied for this type of testing. It is obvious that the developer is very close to this type of testing. Often, developers use white-box testing techniques to test their own design and code. This testing is also known as *structural* or *development testing*.

In white-box testing, structure means the logic of the program which has been implemented in the language code. The intention is to test this logic so that required results or functionalities can be achieved. Thus, white-box testing ensures that the internal parts of the software are adequately tested.

OBJECTIVES

After reading this chapter, you should be able to understand:

- White-box testing demands complete understanding of the program logic/structure
- Test case designing using white-box testing techniques
- Basis path testing method
- Building a path testing tool using graph matrices
- Loop testing
- Data flow testing method
- Mutation testing method

5.1 Need of White-box testing

Is white-box testing really necessary? Can't we write the code and simply test the software using black-box testing techniques? The supporting reasons for white-box testing are given below:

1. In fact, white-box testing techniques are used for testing the module for initial stage testing. Black-box testing is the second stage for testing the software. Though test cases for black box can be designed earlier than white-box test cases, they cannot be executed until the code is produced and checked using white-box testing techniques. Thus, white-box testing is not an alternative but an essential stage.
2. Since white-box testing is complementary to black-box testing, there are categories of bugs which can be revealed by white-box testing, but

not through black-box testing. There may be portions in the code which are not checked when executing functional test cases, but these will be executed and tested by white-box testing.

3. Errors which have come from the design phase will also be reflected in the code, therefore we must execute white-box test cases for verification of code (unit verification).
4. We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis. White-box testing explores these paths too.
5. Some typographical errors are not observed and go undetected and are not covered by black-box testing techniques. White-box testing techniques help detect these errors.

5.2 Logic Coverage Criteria

Structural testing considers the program code, and test cases are designed based on the logic of the program such that every element of the logic is covered. Therefore the intention in white-box testing is to cover the whole logic. Discussed below are the basic forms of logic coverage.

Statement Coverage

The first kind of logic coverage can be identified in the form of statements. It is assumed that if all the statements of the module are executed once, every bug will be notified.

Consider the following code segment shown in Fig. 5.1.

```
scanf ("%d", &x);
scanf ("%d", &y);
while (x != y)
{
     if (x > y)
          x = x - y;
     else
          y = y - x;
 }
 printf ("x = ", x);
 printf ("y = ", y);
```

Figure 5.1 Sample code

If we want to cover every statement in the above code, then the following test cases must be designed:

Test case 1: $x = y = n$, where n is any number

Test case 2: $x = n$, $y = n'$, where n and n' are different numbers.

Test case 1 just skips the while loop and all loop statements are not executed. Considering test case 2, the loop is also executed. However, every statement inside the loop is not executed. So two more cases are designed:

Test case 3: $x > y$

Test case 4: $x < y$

These test cases will cover every statement in the code segment, however statement coverage is a poor criteria for logic coverage. We can see that test case 3 and 4 are sufficient to execute all the statements in the code. But, if we execute only test case 3 and 4, then conditions and paths in test case 1 will never be tested and errors will go undetected. Thus, *statement coverage is a necessary but not a sufficient criteria for logic coverage.*

Decision or Branch Coverage

Branch coverage states that each decision takes on all possible outcomes (True or False) at least once. In other words, each branch direction must be traversed at least once. In the previous sample code shown in Figure 5.1, *while* and *if* statements have two outcomes: True and False. So test cases must be designed such that both outcomes for *while* and *if* statements are tested. The test cases are designed as:

Test case 1: $x = y$

Test case 2: $x \mathrel{!}= y$

Test case 3: $x < y$

Test case 4: $x > y$

Condition Coverage

Condition coverage states that each condition in a decision takes on all possible outcomes at least once. For example, consider the following statement:

while ((I ≤5) && (J < COUNT))

In this loop statement, two conditions are there. So test cases should be designed such that both the conditions are tested for True and False outcomes. The following test cases are designed:

Test case 1: $I \leq 5, J <$ COUNT

Test case 2: $I < 5, J >$ COUNT

Decision/condition Coverage

Condition coverage in a decision does not mean that the decision has been covered. If the decision

if (A && B)

is being tested, the condition coverage would allow one to write two test cases:

Test case 1: *A* is True, *B* is False.

Test case 2: *A* is False, *B* is True.

But these test cases would not cause the THEN clause of the IF to execute (i.e. execution of decision). The obvious way out of this dilemma is a criterion called decision/condition coverage. It requires sufficient test cases such that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each point of entry is invoked at least once [2].

Multiple condition coverage In case of multiple conditions, even decision/condition coverage fails to exercise all outcomes of all conditions. The reason is that we have considered all possible outcomes of each condition in the decision, but we have not taken all combinations of different multiple conditions. Certain conditions mask other conditions. For example, if an AND condition is False, none of the subsequent conditions in the expression will be evaluated. Similarly, if an OR condition is True, none of the subsequent conditions will be evaluated. Thus, condition coverage and decision/condition coverage need not necessarily uncover all the errors.

Therefore, multiple condition coverage requires that we should write sufficient test cases such that all possible combinations of condition outcomes in each decision and all points of entry are invoked at least once. Thus, as in decision/condition coverage, all possible combinations of multiple conditions should be considered. The following test cases can be there:

Test case 1: $A =$ True, $B =$ True

Test case 2: $A =$ True, $B =$ False

Test case 3: $A =$ False, $B =$ True

Test case 4: $A =$ False, $B =$ False

5.3 Basis Path Testing

Basis path testing is the oldest structural testing technique. The technique is based on the control structure of the program. Based on the control structure, a flow graph is prepared and all the possible paths can be covered and executed during testing. Path coverage is a more general criterion as compared to other coverage criteria and useful for detecting more errors. But the problem with path criteria is that programs that contain loops may have an infinite number of possible paths and it is not practical to test all the paths. Some criteria should be devised such that selected paths are executed for maximum

coverage of logic. Basis path testing is the technique of selecting the paths that provide a basis set of execution paths through the program.

The guidelines for effectiveness of path testing are discussed below:

1. Path testing is based on control structure of the program for which flow graph is prepared.
2. Path testing requires complete knowledge of the program's structure.
3. Path testing is closer to the developer and used by him to test his module.
4. The effectiveness of path testing gets reduced with the increase in size of software under test [9].
5. Choose enough paths in a program such that maximum logic coverage is achieved.

5.3.1 Control Flow Graph

The control flow graph is a graphical representation of control structure of a program. Flow graphs can be prepared as a directed graph. A directed graph (V, E) consists of a set of vertices V and a set of edges E that are ordered pairs of elements of V. Based on the concepts of directed graph, following notations are used for a flow graph:

- ***Node*** It represents one or more procedural statements. The nodes are denoted by a circle. These are numbered or labeled.
- ***Edges or links*** They represent the flow of control in a program. This is denoted by an arrow on the edge. An edge must terminate at a node.
- ***Decision node*** A node with more than one arrow leaving it is called a decision node.
- ***Junction node*** A node with more than one arrow entering it is called a junction.
- ***Regions*** Areas bounded by edges and nodes are called regions. When counting the regions, the area outside the graph is also considered a region.

5.3.2 Flow Graph Notations for Different Programming Constructs

Since a flow graph is prepared on the basis of control structure of a program, some fundamental graphical notations are shown here (see Fig. 5.2) for basic programming constructs.

Using the above notations, a flow graph can be constructed. Sequential statements having no conditions or loops can be merged in a single node. That is why, the flow graph is also known as *decision-to-decision-graph* or *DD graph*.

Figure 5.2

5.3.3 Path Testing Terminology

Path A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit. A path may go through several junctions, processes, or decisions, one or more times.

Segment Paths consist of segments. The smallest segment is a link, that is, a single process that lies between two nodes (e.g., junction-process-junction, junction-process-decision, decision-process-junction, decision-process-decision). A direct connection between two nodes, as in an unconditional GOTO, is also called a process by convention, even though no actual processing takes place.

Path segment A path segment is a succession of consecutive links that belongs to some path.

Length of a path The length of a path is measured by the number of links in it and not by the number of instructions or statements executed along the path. An alternative way to measure the length of a path is by the number of nodes traversed. This method has some analytical and theoretical benefits. If programs are assumed to have an entry and an exit node, then the number of links traversed is just one less than the number of nodes traversed.

Independent path An independent path is any path through the graph that introduces at least one new set of processing statements or new conditions. An independent path must move along at least one edge that has not been traversed before the path is defined [9,28].

5.3.4 Cyclomatic Complexity

McCabe [24] has given a measure for the logical complexity of a program by considering its control flow graph. His idea is to measure the complexity by considering the number of paths in the control graph of the program. But even for simple programs, if they contain at least one cycle, the number of paths is infinite. Therefore, he considers only independent paths.

Figure 5.3 Sample graph

In the graph shown in Figure 5.3, there are six possible paths: *acei, acgh, acfh, bdei, bdgh, bdfj*.

In this case, we would see that, of the six possible paths, only four are independent, as the other two are always a linear combination of the other four paths. Therefore, the number of independent paths is 4. In graph theory, it can be demonstrated that in a strongly connected graph (in which each node can be reached from any other node), the number of independent paths is given by

$$V(G) = e - n + 1$$

where n is the number of nodes and e is the number of arcs/edges.

However, it may be possible that the graph is not strongly connected. In that case, the above formula does not fit. Therefore, to make the graph strongly connected, we add an arc from the last node to the first node of the graph. In this way, the flow graph becomes a strongly connected graph. But by doing this, we increase the number of arcs by 1 and therefore, the number of independent paths (as a function of the original graph) is given by

$$V(G) = e - n + 2$$

This is called the *cyclomatic number* of a program. We can calculate the cyclomatic number only by knowing the number of choice points (decision nodes) d in the program. It is given by

$$V(G) = d + 1$$

This is also known as *Miller's theorem.* We assume that a k-way decision point contributes for $k-1$ choice points.

The program may contain several procedures also. These procedures can be represented as separate flow graphs. These procedures can be called from any point but the connections for calling are not shown explicitly. The cyclomatic number of the whole graph is then given by the sum of the numbers of each graph. It is easy to demonstrate that, if p is the number of graphs and e and n are referred to as the whole graph, the cyclomatic number is given by

$$V(G) = e - n + 2p$$

And Miller's theorem becomes

$$V(G) = d + p$$

Formulae Based on Cyclomatic Complexity

Based on the cyclomatic complexity, the following formulae are being summarized.

Cyclomatic complexity number can be derived through any of the following three formulae

1. $V(G) = e - n + 2p$
   where e is number of edges, n is the number of nodes in the graph, and p is number of components in the whole graph.
2. $V(G) = d + p$
   where d is the number of decision nodes in the graph.
3. $V(G) =$ number of regions in the graph

Calculating the number of decision nodes for Switch-Case/Multiple If-Else When a decision node has exactly two arrows leaving it, then we count it as a single decision node. However, switch-case and multiple if-else statements have more than two arrows leaving a decision node, and in these cases, the formula to calculate the number of nodes is

$d = k - 1$, where k is the number of arrows leaving the node.

Calculating the cyclomatic complexity number of the program having many connected components Let us say that a program P has three components: X, Y, and Z. Then we prepare the flow graph for P and for components, X, Y, and Z. The complexity number of the whole program is

$$V(G) = V(P) + V(X) + V(Y) + V(Z)$$

We can also calculate the cyclomatic complexity number of the full program with the first formula by counting the number of nodes and edges in all the components of the program collectively and then applying the formula

$$V(G) = e - n + 2P$$

The complexity number derived collectively will be same as calculated above. Thus,

$$V(P \cup X \cup Y \cup Z) = V(P) + V(X) + V(Y) + V(Z)$$

Guidelines for Basis Path Testing

We can use the cyclomatic complexity number in basis path testing. Cyclomatic number, which defines the number of independent paths, can be utilized as an upper bound for the number of tests that must be conducted to ensure that all the statements have been executed at least once. Thus, independent paths are prepared according to the upper limit of the cyclomatic number. The set of independent paths becomes the basis set for the flow graph of the program. Then test cases can be designed according to this basis set.

The following steps should be followed for designing test cases using path testing:

- Draw the flow graph using the code provided for which we have to write test cases.
- Determine the cyclomatic complexity of the flow graph.
- Cyclomatic complexity provides the number of independent paths. Determine a basis set of independent paths through the program control structure.
- The basis set is in fact the base for designing the test cases. Based on every independent path, choose the data such that this path is executed.

Example 5.1

Consider the following program segment:

```
main()
{
    int number, index;
1.  printf("Enter a number");
2.  scanf("%d, &number);
3.  index = 2;
4.  while(index <= number - 1)
5.  {
6.      if (number % index == 0)
7.      {
8.          printf("Not a prime number");
9.          break;
10.     }
11.     index++;
12.     }
```

```
13.     if(index == number)
14.         printf("Prime number");
15. } //end main
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all the methods.
(c) List all independent paths.
(d) Design test cases from independent paths.

Solution

(a) DD graph

For a DD graph, the following actions must be done:

- Put the line numbers on the execution statements of the program, as shown in Fig. 5.4. Start numbering the statements after declaring the variables, if no variables have been initialized. Otherwise, start from the statement where a variable has been initialized.

Figure 5.4 DD graph for Example 5.1

- Put the sequential statements in one node. For example, statements 1, 2, and 3 have been put inside one node.
- Put the edges between the nodes according to their flow of execution.
- Put alphabetical numbering on each node like *A*, *B*, etc.

The DD graph of the program is shown in Figure 5.4.

(b) Cyclomatic complexity

(i) $V(G) = e - n + 2 * p$

$= 10 - 8 + 2$

$= 4$

Figure 5.5 DD graph for Example 5.1 showing regions

(ii) $V(G) =$ Number of predicate nodes + 1

$= 3$ (Nodes *B*, *C*, and *F*) + 1

$= 4$

(iii) $V(G) = \text{Number of regions}$
$= 4(R1, R2, R3, R4)$

(c) Independent paths

Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph as shown below:

(i) A-B-F-H
(ii) A-B-F-G-H
(iii) A-B-C-E-B-F-G-H
(iv) A-B-C-D-F-H

(d) Test case design from the list of independent paths

Test case ID	Input num	Expected result	Independent paths covered by test case
1	1	No output is displayed	A-B-F-H
2	2	Prime number	A-B-F-G-H
3	4	Not a prime number	A-B-C-D-F-H
4	3	Prime number	A-B-C-E-B-F-G-H

Example 5.2

Consider the following program that reads in a string and then checks the type of each character.

```
    main()
    {
        char string [80];
        int index;
1.      printf("Enter the string for checking its characters");
2.      scanf("%s", string);
3.      for(index = 0; string[index] != '\0'; ++index)   {
4.          if((string[index] >= '0' && (string[index] <='9'
5.                      printf("%c is a digit", string[index]);
6.          else if ((string[index] >= 'A' && string[index] <'Z'))  ||
                    ((string[index] >= 'a' && (string[index] <'z')))
7.                  printf("%c is an alphabet", string[index]);
8.          else
9.                  printf("%c is a special character", string[index]);
10.         }
11. }
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all the methods.
(c) List all independent paths.
(d) Design test cases from independent paths.

Solution

(a) DD graph

The DD graph of the program is shown in Fig. 5.6.

Figure 5.6 DD graph for Example 5.2

(b) Cyclomatic complexity

(i) $V(G) = e - n + 2 * P$
$= 10 - 8 + 2$
$= 4$

(ii) $V(G) =$ Number of predicate nodes + 1
$= 3$ (Nodes B, C) + 1
$= 4$

Node C is a multiple IF-THEN-ELSE, so for finding out the number of predicate nodes for this case, follow the following formula:

Number of predicated nodes

= Number of links out of main node − 1

= 3 – 1 = 2 (For node *C*)

(iii) $V(G)$ = Number of regions

= 4

(c) Independent paths

Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph as shown below:

(i) *A-B-H*

(ii) *A-B-C-D-G-B-H*

(iii) *A-B-C-E-G-B-H*

(iv) *A-B-C-F-G-B-H*

(d) Test case design from the list of independent paths

Test Case ID	Input Line	Expected Output	Independent paths covered by Test case
1	0987	0 is a digit 9 is a digit 8 is a digit 7 is a digit	A-B-C-D-G-B-H A-B-H
2	AzxG	A is a alphabet z is a alphabet x is a alphabet G is a alphabet	A-B-C-E-G-B-H A-B-H
3	@#	@ is a special character # is a special character	A-B-C-F- G-B-H A-B-H

Example 5.3

Consider the following program:

```
     main()
     {
         char chr;
1.       printf ("Enter the special character\n");
2.       scanf (%c", &chr);
3.       if (chr != 48) && (chr != 49) && (chr != 50) && (chr != 51) &&
            (chr != 52) && (chr != 53) && (chr != 54) && (chr != 55) &&
            (chr != 56) && (chr != 57)
4.       {
5.           switch(chr)
```

```
6.          {
7.          Case '*': printf("It is a special character");
8.          break;
9.          Case '#': printf("It is a special character");
10.         break;
11.         Case '@': printf("It is a special character");
12.         break;
13.         Case '!':  printf("It is a special character");
14.         break;
15.         Case '%': printf("It is a special character");
16.         break;
17.         default : printf("You have not entered a special character");
18.         break;
19.         }// end of switch
20.     } // end of If
21.     else
22.         printf("You have not entered a character");
23. } // end of main()
```

(a) Draw the DD graph for the program.

(b) Calculate the cyclomatic complexity of the program using all the methods.

(c) List all independent paths.

(d) Design test cases from independent paths.

Solution

(a) DD graph

The DD graph of the program is shown in Fig. 5.7.

Figure 5.7 DD graph for Example 5.3

(b) Cyclomatic complexity

(i) $V(G) = e - n + 2p$

$= 17 - 12 + 2$

$= 7$

(ii) $V(G) =$ Number of predicate nodes + 1

$= 2$ (Nodes B, C) + 1

$= 7$

Node C is a switch-case, so for finding out the number of predicate nodes for this case, follow the following formula:

Number of predicated nodes

= Number of links out of main node –1

= 6 – 1 = 5 (For node C)

(iii) $V(G) =$ Number of regions = 7

(c) Independent paths

Since the cyclomatic complexity of the graph is 7, there will be 7 independent paths in the graph as shown below:

1. *A-B-D-L*
2. *A-B-C-E-K-L*
3. *A-B-C-F-K-L*
4. *A-B-C-G-K-L*
5. *A-B-C-H-K-L*
6. *A-B-C-I-K-L*
7. *A-B-C-J-K-L*

(d) Test Case Design from the list of Independent Paths

Test Case ID	Input Character	Expected Output	Independent path covered by Test Case
1	(	You have not entered a character	A-B-D-L
2	*	It is a special character	A-B-C-E-K-L
3	#	It is a special character	A-B-C-F-K-L
4	@	It is a special character	A-B-C-G-K-L
5	!	It is a special character	A-B-C-H-K-L
6	%	It is a special character	A-B-C-I-K-L
7	$	You have not entered a special character	A-B-C-J-K-L

Example 5.4

Consider a program to arrange numbers in ascending order from a given list of N numbers.

```
      main()
      {
      int num,small;
1.    int i,j,sizelist,list[10],pos,temp;
      clrscr();
      printf("\nEnter the size of list :\n ");
      scanf("%d",&sizelist);
2.    for(i=0;i<sizelist;i++)
      {
3.    printf("\nEnter the number");
      scanf ("%d",&list[i]);
      }
4.    for(i=0;i<sizelist;i++)
      {
5.    small=list[i];
      pos=i;
6.    for(j=i+1;j<sizelist;j++)
      {
7.    if(small>list[j])
      {
8.    small=list[j];
      pos=j;
      }
9.    }
      temp=list[i];
10.   list[i]=list[pos];
      list[pos]=temp;
11.   }
12.   printf("\nList of the numbers in ascending order : ");
13.   for(i=0;i<sizelist;i++)
14.   printf("\n%d",list[i]);
      getch();
15.   }
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all the methods.
(c) List all independent paths.
(d) Design test cases from independent paths.

Solution

(a) DD graph

The DD graph of the program is shown in Fig. 5.8.

Figure 5.8 DD graph for Example 5.4

(b) Cyclomatic complexity

1. $V(G) = e - n + 2p$

$= 19 - 15 + 2$

$= 6$

2. $V(G) = \text{Number of predicate nodes} + 1$
   $= 5 + 1$
   $= 6$
3. $V(G) = \text{Number of regions}$
   $= 6$

(c) Independent paths

Since the cyclomatic complexity of the graph is 6, there will be 6 independent paths in the graph as shown below:

1. 1-2-3-2-4-5-6-7-8-9-6-10-11-4-12-13-14-13-15
2. 1-2-3-2-4-5-6-7-9-6-10-11-4-12-13-14-13-15
3. 1-2-3-2-4-5-6-10-11-4-12-13-14-13-15
4. 1-2-3-2-4-12-13-14-13-15 (path not feasible)
5. 1-2-4-12-13-15
6. 1-2-3-2-4-12-13-15 (path not feasible)

(d) Test case design from the list of independent paths

Test Case ID	Input	Expected Output	Independent path covered by Test Case
1	Sizelist = 5 List[] = {17,6,7,9,1}	1,6,7,9,17	1
2	Sizelist = 5 List[] = {1,3,9,10,18}	1,3,9,10,18	2
3	Sizelist = 1 List[] = {1}	1	3
4	Sizelist = 0	blank	blank

Example 5.5

Consider the program for calculating the factorial of a number. It consists of main() program and the module fact(). Calculate the individual cyclomatic complexity number for main() and fact() and then, the cyclomatic complexity for the whole program.

```
        main()
        {
            int number;
            int fact();
1.          clrscr();
2.          printf("Enter the number whose factorial is to be found out");
```

```
3.        scanf("%d", &number);
4.        if(number <0)
5.           printf("Facorial cannot be defined for this number);
6.        else
7.           printf("Factorial is %d", fact(number));
8.    }

      int fact(int number)
      {
          int index;
1.        int product =1;
2.        for(index=1; index<=number; index++)
3.           product = product * index;
4.        return(product);
5.    }
```

Solution

DD graph

The DD graph of the program is shown in Fig. 5.9

(a) Flow graph for main () (b) Flow graph for fact ()

Figure 5.9

Cyclomatic complexity of main()

(a) $V(M) = e - n + 2p$

$= 5 - 5 + 2$

$= 2$

(b) $V(M) = d + p$

$= 1 + 1$

$= 2$

(c) $V(M) =$ Number of regions $= 2$

Cyclomatic complexity of fact()

(a) $V(R) = e - n + 2p$
$= 4 - 4 + 2$
$= 2$

(b) $V(R) =$ Number of predicate nodes + 1
$= 1 + 1$
$= 2$

(c) $V(R) =$ Number of regions $= 2$

Cyclomatic complexity of the whole graph considering the full program

(a) $V(G) = e - n + 2p$
$= 9 - 9 + 2 \times 2$
$= 4$
$= V(M) + V(R)$

(b) $V(G) = d + p$
$= 2 + 2$
$= 4$
$= V(M) + V(R)$

(c) $V(G) =$ Number of regions
$= 4$
$= V(M) + V(R)$

5.3.5 Applications of Path Testing

Path testing has been found better suitable as compared to other testing methods. Some of its applications are discussed below.

Thorough testing / More coverage Path testing provides us the best code coverage, leading to a thorough testing. Path coverage is considered better as compared to statement or branch coverage methods because the basis path set provides us the number of test cases to be covered which ascertains the number of test cases that must be executed for full coverage. Generally, branch coverage or other criteria gives us less number of test cases as compared to path testing. Cyclomatic complexity along with basis path analysis employs more comprehensive scrutiny of code structure and control flow, providing a far superior coverage technique.

Unit testing Path testing is mainly used for structural testing of a module. In unit testing, there are chances of errors due to interaction of decision outcomes or control flow problems which are hidden with branch testing. Since each decision outcome is tested independently, path testing uncovers these errors in module testing and prepares them for integration.

Integration testing Since modules in a program may call other modules or be called by some other module, there may be chances of interface errors during calling of the modules. Path testing analyses all the paths on the interface and explores all the errors.

Maintenance testing Path testing is also necessary with the modified version of the software. If you have earlier prepared a unit test suite, it should be run on the modified software or a selected path testing can be done as a part of regression testing. In any case, path testing is still able to detect any security threats on the interface with the called modules.

Testing effort is proportional to complexity of the software Cyclomatic complexity number in basis path testing provides the number of tests to be executed on the software based on the complexity of the software. It means the number of tests derived in this way is directly proportional to the complexity of the software. Thus, path testing takes care of the complexity of the software and then derives the number of tests to be carried out.

Basis path testing effort is concentrated on error-prone software Since basis path testing provides us the number of tests to be executed as a measure of software cyclomatic complexity, the cyclomatic number signifies that the testing effort is only on the error-prone part of the software, thus minimizing the testing effort.

5.4 Graph Matrices

Flow graph is an effective aid in path testing as seen in the previous section. However, path tracing with the use of flow graphs may be a cumbersome and time-consuming activity. Moreover, as the size of graph increases, manual path tracing becomes difficult and leads to errors. A link can be missed or covered twice. So the idea is to develop a software tool which will help in basis path testing.

Graph matrix, a data structure, is the solution which can assist in developing a tool for automation of path tracing. The reason being the properties of graph matrices are fundamental to test tool building. Moreover, testing theory

can be explained on the basis of graphs. Graph theorems can be proved easily with the help of graph matrices. So graph matrices are very useful for understanding the testing theory.

5.4.1 Graph Matrix

A graph matrix is a square matrix whose rows and columns are equal to the number of nodes in the flow graph. Each row and column identifies a particular node and matrix entries represent a connection between the nodes.

The following points describe a graph matrix:

- Each cell in the matrix can be a direct connection or link between one node to another node.
- If there is a connection from node 'a' to node 'b', then it does not mean that there is connection from node 'b' to node 'a'.
- Conventionally, to represent a graph matrix, digits are used for nodes and letter symbols for edges or connections.

Example 5.6

Consider the above graph and represent it in the form of a graph matrix.

Solution

The graph matrix is shown below.

	1	2	3	4
1	a	b	c	
2				d
3				e
4				

Example 5.7

Consider the graph and represent it in the form of a graph matrix.

Solution

The graph matrix is shown below.

	1	2	3	4
1		a+b	c	
2				
3				d
4				

5.4.2 Connection Matrix

Till now, we have learnt how to represent a flow graph into a matrix representation. But this matrix is just a tabular representation and does not provide any useful information. If we add link weights to each cell entry, then graph matrix can be used as a powerful tool in testing. The links between two nodes are assigned a link weight which becomes the entry in the cell of matrix. The link weight provides information about control flow.

In the simplest form, when the connection exists, then the link weight is 1, otherwise 0 (But 0 is not entered in the cell entry of matrix to reduce the complexity). A matrix defined with link weights is called a connection matrix. The connection matrix for Example 5.6 is shown below.

	1	2	3	4
1	1	1	1	
2				1
3				1
4				

The connection matrix for Example 5.7 is shown below.

	1	2	3	4
1		1	1	
2				
3				1
4				

5.4.3 Use of Connection Matrix in Finding Cyclomatic Complexity Number

Connection matrix is used to see the control flow of the program. Further, it is used to find the cyclomatic complexity number of the flow graph. Given below is the procedure to find the cyclomatic number from the connection matrix:

Step 1: For each row, count the number of 1s and write it in front of that row.

Step 2: Subtract 1 from that count. Ignore the blank rows, if any.

Step 3: Add the final count of each row.

Step 4: Add 1 to the sum calculated in Step 3.

Step 5: The final sum in Step 4 is the cyclomatic number of the graph.

The cyclomatic number calculated from the connection matrix of Example 5.6 is shown below.

	1	2	3	4	
1	1	1	1		**3 – 1 = 2**
2				1	**1 – 1 = 0**
3				1	**1 – 1 = 0**
4					
	Cyclomatic number = 2+1 = 3				

The cyclomatic number calculated from the connection matrix of Example 5.7 is shown below.

	1	2	3	4	
1		1	1		**2 – 1 = 1**
2					
3				1	**1 – 1 = 0**
4					
	Cyclomatic number = 1+1 = 2				

5.4.4 Use of Graph Matrix for Finding Set of all Paths

Another purpose of developing graph matrices is to produce a set of all paths between all nodes. It may be of interest in path tracing to find *k*-link paths from one node. For example, how many 2-link paths are there from one node to another node? This process is done for every node resulting in the set of all paths. This set can be obtained with the help of matrix operations. The main objective is to use matrix operations to obtain the set of all paths between all nodes. The set of all paths between all nodes is easily expressed in terms of matrix operations.

The power operation on matrix expresses the relation between each pair of nodes via intermediate nodes under the assumption that the relation is transitive (mostly, relations used in testing are transitive). For example, the square of matrix represents path segments that are 2-links long. Similarly, the cube power of matrix represents path segments that are 3-links long.

Generalizing, we can say that mth power of the matrix represents path segments that are m-links long.

Example 5.8

Consider the graph matrix in Example 5.6 and find 2-link paths for each node.

Solution

For finding 2-link paths, we should square the matrix. Squaring the matrix yields a new matrix having 2-link paths.

$$\begin{pmatrix} a & b & c & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & e \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} a & b & c & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & e \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} a^2 & ab & ac & bd+ce \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

The resulting matrix shows all the 2-link paths from one node to another. For example, from node 1 to node 2, there is one 2-link, i.e., *ab*.

Example 5.9

Consider the following graph. Derive its graph matrix and find 2-link and 3-link set of paths.

Solution

The graph matrix of the graph is shown below.

$$\begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

First we find 2-link set of paths by squaring this matrix as shown below:

$$\begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & bd & ac & ae+bf \\ 0 & cd & 0 & cf \\ 0 & 0 & dc & de \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Next, we find 3-link set of paths by taking the cube of matrix as shown below:

$$\begin{pmatrix} 0 & bd & ac & ae+bf \\ 0 & cd & 0 & cf \\ 0 & 0 & dc & de \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & acd & bdc & bde+acf \\ 0 & 0 & cdc & cde \\ 0 & dcd & 0 & dcf \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

It can be generalized that for n number of nodes, we can get the set of all paths of $(n-1)$ links length with the use of matrix operations. These operations can be programmed and can be utilized as a software testing tool.

5.5 Loop Testing

Loop testing can be viewed as an extension to branch coverage. Loops are important in the software from the testing viewpoint. If loops are not tested

properly, bugs can go undetected. This is the reason that loops are covered in this section exclusively. Loop testing can be done effectively while performing development testing (unit testing by the developer) on a module. Sufficient test cases should be designed to test every loop thoroughly.

There are four different kinds of loops. How each kind of loop is tested, is discussed below.

Simple loops Simple loops mean, we have a single loop in the flow, as shown in Fig. 5.9.

Figure 5.9 (a)

Figure 5.9 (b)

The following test cases should be considered for simple loops while testing them [9]:

- Check whether you can bypass the loop or not. If the test case for bypassing the loop is executed and, still you enter inside the loop, it means there is a bug.
- Check whether the loop control variable is negative.
- Write one test case that executes the statements inside the loop.
- Write test cases for a typical number of iterations through the loop.
- Write test cases for checking the boundary values of the maximum and minimum number of iterations defined (say min and max) in the loop. It means we should test for min, min+1, min−1, max−1, max, and max+1 number of iterations through the loop.

Nested loops When two or more loops are embedded, it is called a nested loop, as shown in Fig. 5.10. If we have nested loops in the program, it becomes difficult to test. If we adopt the approach of simple tests to test the nested loops, then the number of possible test cases grows geometrically. Thus, the strategy is to start with the innermost loops while holding outer loops to their

minimum values. Continue this outward in this manner until all loops have been covered [9].

Figure 5.10 Nested loops

Concatenated loops The loops in a program may be concatenated (Fig. 5.11). Two loops are concatenated if it is possible to reach one after exiting the other, while still on a path from entry to exit. If the two loops are not on the same path, then they are not concatenated. The two loops on the same path may or may not be independent. If the loop control variable for one loop is used for another loop, then they are concatenated, but nested loops should be treated like nested only.

Figure 5.11 Concatenated loops

Unstructured loops This type of loops is really impractical to test and they must be redesigned or at least converted into simple or concatenated loops.

5.6 Data Flow Testing

In path coverage, the stress was to cover a path using statement or branch coverage. However, data and data integrity is as important as code and code integrity of a module. We have checked every possibility of the control flow of a module. But what about the data flow in the module? Has every data object been initialized prior to use? Have all defined data objects been used for something? These questions can be answered if we consider data objects in the control flow of a module.

Data flow testing is a white-box testing technique that can be used to detect improper use of data values due to coding errors. Errors may be unintentionally introduced in a program by programmers. For instance, a programmer might use a variable without defining it. Moreover, he may define a variable, but not initialize it and then use that variable in a predicate. For example,

```
int a;
if(a == 67) { }
```

In this way, data flow testing gives a chance to look out for inappropriate data definition, its use in predicates, computations, and termination. It identifies potential bugs by examining the patterns in which that piece of data is used. For example, if an out-of-scope data is being used in a computation, then it is a bug. There may be several patterns like this which indicate data anomalies.

To examine the patterns, the control flow graph of a program is used. This test strategy selects the paths in the module's control flow such that various sequences of data objects can be chosen. The major focus is on the points at which the data receives values and the places at which the data initialized has been referenced. Thus, we have to choose enough paths in the control flow to ensure that every data is initialized before use and all the defined data have been used somewhere. Data flow testing closely examines the state of the data in the control flow graph, resulting in a richer test suite than the one obtained from control flow graph based path testing strategies like branch coverage, all statement coverage, etc.

5.6.1 State of a Data Object

A data object can be in the following states:

- ***Defined (d)*** A data object is called *defined* when it is initialized, i.e. when it is on the left side of an assignment statement. Defined state can also be used to mean that a file has been opened, a dynamically allocated object has been allocated, something is pushed onto the stack, a record written, and so on [9].

- ***Killed/Undefined/Released (k)*** When the data has been reinitialized or the scope of a loop control variable finishes, i.e. exiting the loop or memory is released dynamically or a file has been closed.
- ***Usage (u)*** When the data object is on the right side of assignment or used as a control variable in a loop, or in an expression used to evaluate the control flow of a case statement, or as a pointer to an object, etc. In general, we say that the usage is either *computational use* (c-use) or *predicate use* (p-use).

5.6.2 Data-Flow Anomalies

Data-flow anomalies represent the patterns of data usage which may lead to an incorrect execution of the code. An anomaly is denoted by a two-character sequence of actions. For example, 'dk' means a variable is defined and killed without any use, which is a potential bug. There are nine possible two-character combinations out of which only four are data anomalies, as shown in Table 5.1.

Table 5.1 Two-character data-flow anomalies

Anomaly	Explanation	Effect of Anomaly
du	Define-use	Allowed. Normal case.
dk	**Define-kill**	**Potential bug. Data is killed without use after definition.**
ud	Use-define	Data is used and then redefined. Allowed. Usually not a bug because the language permits reassignment at almost any time.
uk	Use-kill	Allowed. Normal situation.
ku	**Kill-use**	**Serious bug because the data is used after being killed.**
kd	Kill-define	Data is killed and then redefined. Allowed.
dd	**Define-define**	**Redefining a variable without using it. Harmless bug, but not allowed.**
uu	Use-use	Allowed. Normal case.
kk	**Kill-kill**	**Harmless bug, but not allowed.**

It can be observed that not all data-flow anomalies are harmful, but most of them are suspicious and indicate that an error can occur. In addition to the above two-character data anomalies, there may be single-character data anomalies also. To represent these types of anomalies, we take the following conventions:

$\sim x$: indicates all prior actions are not of interest to x.

$x\sim$: indicates all post actions are not of interest to x.

All single-character data anomalies are listed in Table 5.2.

Table 5.2 Single-character data-flow anomalies

Anomaly	Explanation	Effect of Anomaly
~d	First definition	Normal situation. Allowed.
~u	**First Use**	**Data is used without defining it. Potential bug.**
~k	**First Kill**	**Data is killed before defining it. Potential bug.**
D~	**Define last**	**Potential bug.**
U~	Use last	Normal case. Allowed.
K~	Kill last	Normal case. Allowed.

5.6.3 Terminology Used in Data Flow Testing

In this section, some terminology [9,20], which will help in understanding all the concepts related to data-flow testing, is being discussed. Suppose P is a program that has a graph $G(P)$ and a set of variables V. The graph has a single entry and exit node.

Definition node Defining a variable means assigning value to a variable for the very first time in a program. For example, input statements, assignment statements, loop control statements, procedure calls, etc.

Usage node It means the variable has been used in some statement of the program. Node n that belongs to $G(P)$ is a usage node of variable v, if the value of variable v is used at the statement corresponding to node n. For example, output statements, assignment statements (right), conditional statements, loop control statements, etc.

A usage node can be of the following two types:

(i) Predicate Usage Node: If usage node n is a predicate node, then n is a predicate usage node.

(ii) Computation Usage Node: If usage node n corresponds to a computation statement in a program other than predicate, then it is called a computation usage node.

Loop-free path segment It is a path segment for which every node is visited once at most.

Simple path segment It is a path segment in which at most one node is visited twice. A simple path segment is either loop-free or if there is a loop, only one node is involved.

Definition-use path (du-path) A du-path with respect to a variable v is a path between the definition node and the usage node of that variable. Usage node can either be a p-usage or a c-usage node.

Definition-clear path(dc-path) A dc-path with respect to a variable *v* is a path between the definition node and the usage node such that no other node in the path is a defining node of variable *v*.

The du-paths which are not dc-paths are important from testing viewpoint, as these are potential problematic spots for testing persons. Those du-paths which are definition-clear are easy to test in comparison to du-paths which are not dc-paths. The application of data flow testing can be extended to debugging where a testing person finds the problematic areas in code to trace the bug. So the du-paths which are not dc-paths need more attention.

5.6.4 Static Data Flow Testing

With static analysis, the source code is analysed without executing it. Let us consider an example of an application given below.

Example 5.10

Consider the program given below for calculating the gross salary of an employee in an organization. If his basic salary is less than Rs 1500, then HRA = 10% of basic salary and DA = 90% of the basic. If his salary is either equal to or above Rs 1500, then HRA = Rs 500 and DA = 98% of the basic salary. Calculate his gross salary.

```
    main()
    {
1.      float bs, gs, da, hra = 0;
2.      printf("Enter basic salary");
3.      scanf("%f", &bs);
4.      if(bs < 1500)
5.      {
6.          hra = bs * 10/100;
7.          da = bs * 90/100;
8.      }
9.      else
10.     {
11.         hra = 500;
12.         da = bs * 98/100;
13.     }
14.     gs = bs + hra + da;
15.     printf("Gross Salary = Rs. %f", gs);
16. }
```

Find out the define-use-kill patterns for all the variables in the source code of this application.

Solution

For variable 'bs', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	3	Normal case. Allowed
du	3-4	Normal case. Allowed
uu	4-6, 6-7, 7-12, 12-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

For variable 'gs', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	14	Normal case. Allowed
du	14-15	Normal case. Allowed
uk	15-16	Normal case. Allowed
K~	16	Normal case. Allowed

For variable 'da', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	7	Normal case. Allowed
du	7-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

For variable 'hra', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	1	Normal case. Allowed
dd	1-6 or 1-11	Double definition. Not allowed. Harmless bug.
du	6-14 or 11-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

From the above static analysis, it was observed that static data flow testing for the variable 'hra' discovered one bug of double definition in line number 1.

Static Analysis is not Enough

It is not always possible to determine the state of a data variable by just static analysis of the code. For example, if the data variable in an array is used as an index for a collection of data elements, we cannot determine its state by static analysis. Or it may be the case that the index is generated dynamically during execution, therefore we cannot guarantee what the state of the array element is referenced by that index. Moreover, the static data-flow testing might denote a certain piece of code to be anomalous which is never executed and hence, not completely anomalous. Thus, all anomalies using static analysis cannot be determined and this problem is provably unsolvable.

5.6.5 Dynamic Data Flow Testing

Dynamic data flow testing is performed with the intention to uncover possible bugs in data usage during the execution of the code. The test cases are designed in such a way that every definition of data variable to each of its use is traced and every use is traced to each of its definition. Various strategies are employed for the creation of test cases. All these strategies are defined below.

All-du Paths (ADUP) It states that every du-path from every definition of every variable to every use of that definition should be exercised under some test. It is the strongest data flow testing strategy, since it is a superset of all other data flow testing strategies. Moreover, this strategy requires the maximum number of paths for testing.

All-uses (AU) This states that for every use of the variable, there is a path from the definition of that variable (nearest to the use in backward direction) to the use.

All-p-uses/Some-c-uses (APU + C) This strategy states that for every variable and every definition of that variable, include at least one dc-path from the definition to every predicate use. If there are definitions of the variable with no p-use following it, then add computational use (c-use) test cases as required to cover every definition.

All-c-uses/Some-p-uses (ACU + P) This strategy states that for every variable and every definition of that variable, include at least one dc-path from the

definition to every computational use. If there are definitions of the variable with no c-use following it, then add predicate use (c-use) test cases as required to cover every definition.

All-Predicate-Uses (APU) It is derived from the APU+C strategy and states that for every variable, there is a path from every definition to every p-use of that definition. If there is a definition with no p-use following it, then it is dropped from contention.

All-Computational-Uses (ACU) It is derived from the strategy ACU+P strategy and states that for every variable, there is a path from every definition to every c-use of that definition. If there is a definition with no c-use following it, then it is dropped from contention.

All-Definition (AD) It states that every definition of every variable should be covered by at least one use of that variable, be that a computational use or a predicate use.

Example 5.11

Consider the program given below. Draw its control flow graph and data flow graph for each variable used in the program, and derive data flow testing paths with all the strategies discussed above.

```
    main()
    {
    int work;
 0. double payment =0;
 1. scanf("%d", work);
 2. if (work > 0) {
 3.     payment = 40;
 4. if (work > 20)
 5. {
 6.     if(work <= 30)
 7.         payment = payment + (work - 25) * 0.5;
 8.     else
 9.     {
10.         payment = payment + 50 + (work -30) * 0.1;
11.             if (payment >= 3000)
12.                 payment = payment * 0.9;
13.     }
14. }
15. }
16. printf("Final payment", payment);
```

Solution

Figure 5.12 shows the control flow graph for the given program.

Figure 5.12 DD graph for Example 5.11

Figure 5.13 shows the data flow graph for the variable 'payment'.

Figure 5.13 Data flow graph for 'payment'

Figure 5.14 shows the data flow graph for the variable 'work'.

Figure 5.14 Data flow graph for variable 'work'

Prepare a list of all the definition nodes and usage nodes for all the variables in the program.

Variable	Defined At	Used At
Payment	0,3,7,10,12	7,10,11,12,16
Work	1	2,4,6,7,10

Data flow testing paths for each variable are shown in Table 5.3.

Table 5.3 Data flow testing paths

Strategy	Payment	Work
All Uses(AU)	3-4-5-6-7 10-11 10-11-12 12-13-14-15-16 3-4-5-6-8-9-10	1-2 1-2-3-4 1-2-3-4-5-6 1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10
All p-uses (APU)	0-1-2-3-4-5-6-8-9-10-11	1-2 1-2-3-4 1-2-3-4-5-6

All c-uses (ACU)	0-1-2-16 3-4-5-6-7 3-4-5-6-8-9-10 3-4-15-16 7-14-15-16 10-11-12 10-11-13-14-15-16 12-13-14-15-16	1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10
All-p-uses / Some-c-uses (APU + C)	0-1-2-3-4-5-6-8-9-10-11 10-11-12 12-13-14-15-16	1-2 1-2-3-4 1-2-3-4-5-6 1-2-3-4-5-6-8-9-10
All-c-uses / Some-p-uses (ACU + P)	0-1-2-16 3-4-5-6-7 3-4-5-6-8-9-10 3-4-15-16 7-14-15-16 10-11-12 10-11-13-14-15-16 12-13-14-15-16 0-1-2-3-4-5-6-8-9-10-11	1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10 1-2-3-4-5-6
All-du-paths (ADUP)	0-1-2-3-4-5-6-8-9-10-11 0-1-2-16 3-4-5-6-7 3-4-5-6-8-9-10 3-4-15-16 7-14-15-16 10-11-12 10-11-13-14-15-16 12-13-14-15-16	1-2 1-2-3-4 1-2-3-4-5-6 1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10
All Definitions (AD)	0-1-2-16 3-4-5-6-7 7-14-15-16 10-11 12-13-14-15-16	1-2

5.6.6 Ordering of Data Flow Testing Strategies

While selecting a test case, we need to analyse the relative strengths of various data flow testing strategies. Figure 5.15 depicts the relative strength of the data flow strategies. In this figure, the relative strength of testing strategies reduces

along the direction of the arrow. It means that all-du-paths (ADPU) is the strongest criterion for selecting the test cases.

Figure 5.15 Data-flow testing strategies

5.7 Mutation Testing

Mutation testing is the process of mutating some segment of code (putting some error in the code) and then, testing this mutated code with some test data. If the test data is able to detect the mutations in the code, then the test data is quite good, otherwise we must focus on the quality of test data. Therefore, mutation testing helps a user create test data by interacting with the user to iteratively strengthen the quality of test data.

During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains one fault. Test data are used to execute these faulty programs with the goal of causing each faulty program to fail. Faulty programs are called *mutants* of the original program and a mutant is said to be *killed* when a test case causes it to fail. When this happens, the mutant is considered *dead* and no longer needs to remain in the testing process, since the faults represented by that mutant have been detected, and more importantly, it has satisfied its requirement of identifying a useful test case. Thus, the main objective is to select efficient test data which have error-detection power. The criterion for this test data is to differentiate the initial program from the mutant. This distinguish-ability between the initial program and its mutant will be based on test results.

5.7.1 Primary Mutants

Let us take an example of a C program to understand primary mutants.

```
...
if (a > b)
    x = x + y;
else
    x = y;
printf("%d", x);
...
```

We can consider the following mutants for the above example:

M1: `x = x - y;`
M2: `x = x / y;`
M3: `x =  x + 1;`
M4: `printf("%d", y);`

When the mutants are single modifications of the initial program using some operators as shown above, they are called *primary mutants*. Mutation operators are dependent on programming languages. Note that each of the mutated statements represents a separate program. The results of the initial program and its mutants are shown below.

Test Data	x	y	Initial Program Result	Mutant Result
TD1	2	2	4	0 (M1)
TD2(x and y # 0)	4	3	7	1.4 (M2)
TD3 (y #1)	3	2	5	4 (M3)
TD4(y #0)	5	2	7	2 (M4)

5.7.2 Secondary Mutants

Let us take another example program as shown below:

```
if (a < b)
    c = a;
```

Now, mutants for this code may be as follows:

```
M1 : if (a <= b-1)
                c = a;
M2: if (a+1 <= b)
                c = a;
M3: if (a == b)
                c = a+1;
```

This class of mutants is called *secondary mutants* when multiple levels of mutation are applied on the initial program. In this case, it is very difficult to identify the initial program from its mutants.

Example 5.12

Consider the program *P* shown below.

```
r = 1;
for (i = 2; i<=3; ++i) {
    if (a[i] > a[r]
        r = i;
}
```

The mutants for *P* are:

M1:

```
r = 1;
for (i = 1; i<=3; ++i) {
    if (a[i] > a[r])
        r = i;
}
```

M2:

```
r = 1;
for (i = 2; i<=3; ++i) {
    if (i > a[r])
        r = i;
}
```

M3:

```
r = 1;
for (i = 2; i<=3; ++i) {
    if (a[i] >= a[r])
        r = i;
}
```

M4:

```
r = 1;
for (i = 1; i<=3; ++i) {
    if (a[r] > a[r])
        r = i;
}
```

Let us consider the following test data selection:

	a[1]	a[2]	a[3]
TD1	1	2	3
TD2	1	2	1
TD3	3	1	2

We apply these test data to mutants, M1, M2, M3, and M4.

	P	M1	M2	M3	M4	Killed Mutants
TD1	3	3	3	3	1	M4
TD2	2	2	3	2	1	M2 and M4
TD3	1	1	1	1	1	none

We need to look at the efficiency of the proposed test data. It can be seen that M1 and M3 are not killed by the test data selection. Therefore, the test data is incomplete. Therefore, we need to add a new test data, TD4 = {2,2,1}, then this test data kills M3.

5.7.3 Mutation Testing Process

The mutation testing process is discussed below:

- Construct the mutants of a test program.
- Add test cases to the mutation system and check the output of the program on each test case to see if it is correct.
- If the output is incorrect, a fault has been found and the program must be modified and the process restarted.
- If the output is correct, that test case is executed against each live mutant.
- If the output of a mutant differs from that of the original program on the same test case, the mutant is assumed to be incorrect and is killed.
- After each test case has been executed against each live mutant, each remaining mutant falls into one of the following two categories.
    - One, the mutant is functionally equivalent to the original program. An equivalent mutant always produces the same output as the original program, so no test case can kill it.
    - Two, the mutant is killable, but the set of test cases is insufficient to kill it. In this case, new test cases need to be created, and the process iterates until the test set is strong enough to satisfy the tester.

- The mutation score for a set of test data is the percentage of non-equivalent mutants killed by that data. If the mutation score is 100%, then the test data is called *mutation-adequate*.

SUMMARY

White-box testing is a dynamic testing category under which the software is tested using the structure or logic of the program under test. This technique is closer to a developer who needs to test his module while developing it. The developer cannot design and code the module in one go and therefore he is supposed to verify his logic using various test case design methods.

White-box testing is necessary, as it is one of the steps to verify the correctness of the program and consequently in enhancing the quality of the program. All the methods of white-box testing have been discussed in this chapter, taking sufficient number of examples such that a new method can be learnt easily and the reader is ready to design the test cases using a particular method.

Let us review the important concepts described in this chapter:

- White-box testing is the technique wherein it is required to understand the structure and logic of the program to test the software.
- White-box testing is largely done by the developer of the software.
- White-box testing covers the logic of the program. The forms of logic coverage are: statement coverage, decision/branch coverage, condition coverage, decision-condition coverage, and multiple condition.
- Basis path testing is the technique of selecting the paths that provide a basis set of execution paths through the program.
- The basis set is derived by calculating the cyclomatic complexity number. Having the knowledge of this number, independent paths are derived from the flow graph of the program. Corresponding to independent paths, test cases are prepared which form the basis set.
- An independent path is any path through the graph that introduces at least one new set of processing statements or new conditions.
- Cyclomatic complexity is the number which tells us about the complexity of the design of a module. It should be less than 10, otherwise the module should be redesigned.
- Cyclomatic complexity of a module can be calculated by the following methods:
  - $V(G) = e - n + 2P$

    where e is number of edges, n is the number of nodes in the graph, and P is the number of components in the whole graph
  - $V(G) = d + P$

    where d is the number of decision nodes in the graph.
  - $V(G)$ = number of regions in the graph
  - Using graph matrices
- Graph matrix, a data structure, can assist in developing a tool for automation of path tracing.

- A graph matrix is a square matrix whose rows and columns are equal to the number of nodes in the flow graph.
- A matrix defined with link weights is called a connection matrix.
- The connection matrix is used to find the cyclomatic complexity number of the flow graph.
- Graph matrices are also used to produce a set of all paths between all nodes. An *m*th power of the matrix represents all path segments *m* links long.
- Data flow testing is a white-box testing technique that can be used to detect improper use of data values due to coding errors.
- Data flow anomalies represent the patterns of data usage which may lead to an incorrect execution of the code.
- Definition node in a data flow graph is the node wherein a variable is assigned a value for the very first time in the program.
- Usage node in a data flow graph is the node wherein the variable has already been used in some statement of the program.
- A du-path with respect to a variable *v* is a path between the definition node and the usage node of that variable.
- A dc-path with respect to a variable *v* is a path between the definition node and the usage node such that no other node in the path is a defining node of variable *v*.
- du-paths which are not definition-clear paths are important from testing viewpoint, as these are potential problematic spots for testing persons. du-paths which are definition-clear are easy to test in comparison to those which are not dc-paths.
- The application of data flow testing can be extended to debugging where a testing person finds the problematic areas in the code to trace the bug. So the du-paths which are not definition-clear need more attention of the tester.
- During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains a fault. Test data are used to execute these faulty programs with the goal of causing each faulty program to fail. Faulty programs are called mutants of the original program and a mutant is said to be killed when a test case causes it to fail.
- When the mutants are single modifications of the initial program using some operators, they are called primary mutants.
- When multiple levels of mutation are applied on the initial program, they are called secondary mutants.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. White-box testing is _____ to black-box testing.
   (a) mutually exclusive
   (b) complementary
   (c) not related

2. The effectiveness of path testing rapidly _____ as the size of software under test.
   (a) decreases
   (b) increases
   (c) does not change
   (d) none of the above
3. A node with more than one arrow leaving it is called a _____
   (a) decision node
   (b) junction node
   (c) region
   (d) all of the above
4. A node with more than one arrow entering it is called a _____
   (a) decision node
   (b) junction node
   (c) region
   (d) all of the above
5. Areas bounded by edges and nodes are called _____
   (a) decision node
   (b) junction node
   (c) region
   (d) all of the above
6. The length of a path is measured by the number of _____
   (a) instructions
   (b) junction nodes
   (c) decision nodes
   (d) links
7. An independent path is any path through the graph that introduces at least _____ new set of processing statements or new conditions.
   (a) 4
   (b) 3
   (c) 1
   (d) 2
8. The number of independent paths is given by _____.
   (a) $V(G) = e - n + 1$
   (b) $V(G) = 2e - n + 1$
   (c) $V(G) = e - n + 2$
   (d) none of the above
9. According to Mill's Theorem, _____
   (a) $V(G) = d + 2P$

(b) $V(G) = d + P$
(c) $V(G) = 2d + P$
(d) None of the above

10. In data flow anomalies, dd is a _____
    (a) serious bug
    (b) normal case
    (c) harmless bug
    (d) none of the above
11. In data flow anomalies, du is a _____
    (a) serious bug
    (b) normal case
    (c) harmless bug
    (d) none of the above
12. In data flow anomalies, ku is a _____
    (a) serious bug
    (b) normal case
    (c) harmless bug
    (d) none of the above
13. In single-character data anomalies, ~d is _____
    (a) potential bug
    (b) normal situation
    (c) none of the above
15. In single-character data anomalies, ~k is _____
    (a) potential bug
    (b) normal situation
    (c) none of the above
16. _____ is the strongest criterion for selecting test cases.
    (a) AD
    (b) APU
    (c) AU
    (d) ADPU

Review Questions

1. What is the need of white-box testing?
2. What are the different criteria for logic coverage?
3. What is basis path testing?

4. Distinguish between decision node and junction node?
5. What is an independent path?
6. What is the significance of cyclomatic complexity?
7. How do you calculate the number of decision nodes for switch-case?
8. How do you calculate the cyclomatic complexity number of the program having many connected components?
9. Consider the program.

```
#include <stdio.h>
main()
{
        int a,b, c,d;
        clrscr();
        printf("enter the two variables a,b");
        scanf("%d %d",&a,&b);
        printf("enter the option 1:Addition,
               2:subtraction,3:multiplication,4:division");
        scanf("%d",&c);
        switch(c)
        {
                case 1:d = a+b;
                printf("Addition of two no.=%d", d);
                break;
                case 2:d = a-b;
                printf("Subtraction of two no.=%d", d);
                break;
                case 3:d = a*b;
                printf("Multiplication of two no.=%d", d);
                break;
                case 4:d = a/b;
                printf("division of two no.=%d",d);
                break;
        }
}
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all four methods.
(c) List all independent paths.
(d) Design all test cases from independent paths.
(e) Derive all du-paths and dc-paths using data flow testing.

10. Consider the program to find the greatest number:

```
#include <stdio.h>
main()
{
        float x,y,z;
        clrscr();
        printf("enter the three variables x,y,z");
        scanf("%f %f %f",&x,&y,&z);
        if(x > y)
        {
            if(x > z)
                printf("x is greatest");
            else
                printf("z is greatest");
        }
        else
        {
            if(y > z)
                printf("y is greatest");
            else
                printf("z is greatest");
        }
    getch();
}
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all four methods.
(c) List all independent paths.
(d) Design all test cases from independent paths.
(e) Derive all du-paths and dc-paths using data flow testing.

11. Consider the following program which multiplies two matrices:

```
#include <stdio.h>
main()
{
int a[SIZE][SIZE], b[SIZE][SIZE], c[SIZE][SIZE], i, j, k, rowl,
colml, row2, colm2;

printf("Enter the order of first matrix <= %d %d \n", SIZE, SIZE);
scanf("%d%d",&row1, colm1);
printf("Enter the order of second matrix <= %d %d \n", SIZE, SIZE);
scanf("%d%d",&row2, colm2);
if(colm1==row2)
```

```
        {
        printf("Enter first matrix");
        for(i=0; i<row1; i++)
                {
                for(j=0; j<colml; j++)
                scanf("%d", &a[i][j]);
                }
        printf("Enter second matrix");
        for(i=0; i<row2; i++)
                {
                for(j=0; j<colm2; j++)
                scanf("%d", &b[i][j]);
                }
        printf("Multiplication of two matrices is");
        for(i=0; i<row1; i++)
                {
                for(j=0; j<colm1; j++)
                        {
                        c[i] [j] = 0;
                        for(k=0; k<row2; k++)
                        c[i][j]+ = a[i][k] + b[k][j];
                        printf("%6d", c[i][j]);
                        }
                }
        }
        else
        {
                printf("Matrix multiplication is not possible");
        }
}
```

(a) Draw the DD graph for the program.

(b) Calculate the cyclomatic complexity of the program using all four methods.

(c) List all independent paths.

(d) Design all test cases from the independent paths.

(e) Derive all du-paths and dc-paths using data flow testing.

12. Consider the following program for finding the prime numbers, their sum, and count:

```
main()
{
        int num, flag, sum, count;
        int CheckPrime(int n);
        sum = count = 0;
        printf("Prime number between 1 and 100 are");
        for(num=1; num<=50; num++)
```

```
        {
                flag = CheckPrime(num);
                if(flag)
                {
                        printf("%d", num);
                        sum+ = num;
                        count++;
                }
        }
        printf("Sum of primes %d", count);
}
int CheckPrime(int n)
{
        int srt, d;
        srt = sqrt(n);
        d = 2;
        while(d <= srt)
        {
                If(n%d == 0)
                break;
                d++;
        }
        if(d > srt)
                return(1);
        else
                return(0);
}
```

(a) Draw the DD graph for the program.
(b) This program consists of main() and one module. Calculate the individual cyclomatic complexity number for both and collectively for the whole program. Show that individual cyclomatic complexity of main() and CheckPrime() and cyclomatic complexity of whole program is equal.
(c) List all independent paths.
(d) Design all test cases from independent paths.
(e) Derive all du-paths and dc-paths using data flow testing.

13. Consider the following program for calculating the grade of a student:

```
main()
{
     char grade;
     int s1, s2, s3, s4, total;
     float average;
     printf("Enter the marks of 4 subjects");
     scanf("%d  %d  %d  %d", &s1,&s2,&s3,&s4);
```

```
        if(s1 < 40 || s2 < 40 || s3 < 40 || s4 < 40)
              printf("Student has failed");
        else
        {
              total = s1 + s2 + s3 + s4;
              average = total/4.0;
              if(average > 80.0)
                    grade = 'A';
              else if(average >= 70.0)
                    grade = 'B';
              else if(average >= 60.0)
                    grade = 'C';
              else if(average >= 50.0)
                    grade = 'D';
              else if(average >= 45.0)
                    grade = 'E';
              else
                    grade = 'F';
              switch(grade)
              {
                    case 'A':  printf("Student has got A grade");
                    case 'B':  printf("Student has got B grade");
                    case 'C':  printf("Student has got C grade");
                    case 'D':  printf("Student has got D grade");
                    case 'E':  printf("Student has got E grade");
                    case 'F':  printf("Student has got F grade");
              }
        }
}
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all four methods.
(c) List all independent paths.
(d) Design all test cases from independent paths.

14. Consider the following graph:

(a) Represent this graph in the form of a graph matrix.
(b) Represent this graph in the form of a connection matrix.
(c) Find 2-link paths and 3-link paths for each node.

15. 'Nested loops are problematic areas for testers.' Comment on this.
16. Give examples of each type of data anomaly.
17. Consider Question 9 and perform static data flow analysis by finding out the define-use-kill patterns for all the variables in the source code.
18. Consider Question 9 and draw data flow graph for each variable used in the program and derive data flow testing paths with all the strategies.
19. What is the difference between primary and secondary mutants?
20. Consider Example 5.10. Find out the possible mutants and check how many of them are killed by a set of test data. Add new test data if required.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

6

Static Testing

OBJECTIVES

After reading this chapter, you should be able to understand:

- Static testing is complementary to dynamic testing
- Static testing also improves the software quality
- Some bugs can be detected only through static testing
- Three types of static testing: Inspection, Walkthroughs, and Reviews
- Inspections are the most widely used technique for static testing which is a formal process to detect the bugs early
- Benefits and effectiveness of inspection process
- Variants of inspection process
- Walkthrough is a less formal and less rigorous method as compared to inspection process
- Review is a higher level technique as compared to inspection or walkthrough, as it also includes management representatives

We have discussed dynamic testing techniques that execute the software being built with a number of test cases. However, it suffers from the following drawbacks:

- Dynamic testing uncovers the bugs at a later stage of SDLC and hence is costly to debug.
- Dynamic testing is expensive and time-consuming, as it needs to create, run, validate, and maintain test cases.
- The efficiency of code coverage decreases with the increase in size of the system.
- Dynamic testing provides information about bugs. However, debugging is not always easy. It is difficult and time-consuming to trace a failure from a test case back to its root cause.
- Dynamic testing cannot detect all the potential bugs.

While dynamic testing is an important aspect of any quality assurance program, it is not a universal remedy. Thus, it alone cannot guarantee defect-free product, nor can it ensure a sufficiently high level of software quality.

In response to the above discussion, static testing is a complimentary technique to dynamic testing technique to acquire higher quality software. Static testing techniques do not execute the software and they do not require the bulk of test cases. This type of testing is also known as non-computer based testing or human testing. All the bugs cannot be caught alone by the dynamic testing technique; static testing reveals the errors which are not shown by dynamic testing. Static testing can be applied for most of the verification activities dis-

cussed earlier. Since verification activities are required at every stage of SDLC till coding, static testing also can be applied at all these phases.

Static testing techniques do not demonstrate that the software is operational or that one function of software is working; rather they check the software product at each SDLC stage for conformance with the required specifications or standards. Requirements, design specifications, test plans, source code, user's manuals, maintenance procedures are some of the items that can be statically tested.

Static testing has proved to be a cost-effective technique of error detection. An empirical comparison between static and dynamic testing [26, 27], proves the effectiveness of static testing. Further, Fagan [28] reported that more than 60% of the errors in a program can be detected using static testing. Another advantage of static testing is that it provides the exact location of a bug, whereas dynamic testing provides no indication of the exact source code location of the bug. In other words, we can say that static testing finds the in-process errors before they become bugs.

Static testing techniques help to produce a better product. Given below are some of the benefits of adopting a static testing approach:

- As defects are found and fixed, the quality of the product increases.
- A more technically correct base is available for each new phase of development.
- The overall software life cycle cost is lower, since defects are found early and are easier and less expensive to fix.
- The effectiveness of the dynamic test activity is increased and less time needs to be devoted for testing the product.
- Immediate evaluation and feedback to the author from his/her peers which will bring about improvements in the quality of future products.

The objectives of static testing can be summarized as follows:

- To identify errors in any phase of SDLC as early as possible
- To verify that the components of software are in conformance with its requirements
- To provide information for project monitoring
- To improve the software quality and increase productivity

Types of Static Testing

Static testing can be categorized into the following types:

- Software inspections
- Walkthroughs
- Technical reviews

6.1 Inspections

Software inspections were first introduced at IBM by Fagan in the early 1970s [43]. These can be used to tackle software quality problems because they allow the detection and removal of defects after each phase of the software development process. Inspection process is an in-process manual examination of an item to detect bugs. It may be applied to any product or partial product of the software development process, including requirements, design and code, project management plan, SQA plan, software configuration plan (SCM plan), risk management plan, test cases, user manual, etc. Inspections are embedded in the process of developing products and are done in the early stages of each product's development.

This process does not require executable code or test cases. With inspection, bugs can be found on infrequently executed paths that are not likely to be included in test cases. Software inspection does not execute the code, so it is machine-independent, requires no target system resources or changes to the program's operational behaviour, and can be used much before the target hardware is available for dynamic testing purposes.

The inspection process is carried out by a group of peers. The group of peers first inspect the product at the individual level. After this, they discuss the potential defects of the product observed in a formal meeting. The second important thing about the inspection process is that it is a formal process of verifying a software product. The documents which can be inspected are SRS, SDD, code, and test plan.

An inspection process involves the interaction of the following elements:

- Inspection steps
- Role for participants
- Item being inspected

The entry and exit criteria are used to determine whether an item is ready to be inspected. Entry criteria mean that the item to be inspected is mature enough to be used. For example, for code inspection, the entry criterion is that the code has been compiled successfully. Exit criterion is that once the item has been given for inspection, it should not be updated, otherwise it will not know how many bugs have been reported and corrected through the inspection process and the whole purpose of the inspection is lost.

6.1.1 Inspection Team

For the inspection process, a minimum of the following four team members are required.

Author/Owner/Producer A programmer or designer responsible for producing the program or document. He is also responsible for fixing defects discovered during the inspection process.

Inspector A peer member of the team, i.e. he is not a manager or supervisor. He is not directly related to the product under inspection and may be concerned with some other product. He finds errors, omissions, and inconsistencies in programs and documents.

Moderator A team member who manages the whole inspection process. He schedules, leads, and controls the inspection session. He is the key person with the responsibility of planning and successful execution of the inspection.

Recorder One who records all the results of the inspection meeting.

6.1.2 Inspection Process

A general inspection process (see Fig. 6.1) has the following stages [29,14]:

Figure 6.1 Inspection process

Planning During this phase, the following is executed:

- The product to be inspected is identified.
- A moderator is assigned.
- The objective of the inspection is stated, i.e. whether the inspection is to be conducted for defect detection or something else. If the objective is defect detection, then the type of defect detection like design error, interface error, code error must be specified. The aim is to define an objective for the meeting so that the effort spent in inspections is properly utilized.

During planning, the moderator performs the following activities:

- Assures that the product is ready for inspection
- Selects the inspection team and assigns their roles
- Schedules the meeting venue and time
- Distributes the inspection material like the item to be inspected, checklists, etc.

Overview In this stage, the inspection team is provided with the background information for inspection. The author presents the rationale for the product, its relationship to the rest of the products being developed, its function and intended use, and the approach used to develop it. This information is necessary for the inspection team to perform a successful inspection.

The opening meeting may also be called by the moderator. In this meeting, the objective of inspection is explained to the team members. The idea is that every member should be familiar with the overall purpose of the inspection.

Individual preparation After the overview, the reviewers individually prepare themselves for the inspection process by studying the documents provided to them in the *overview* session. They point out potential errors or problems found and record them in a log. This log is then submitted to the moderator. The moderator compiles the logs of different members and gives a copy of this compiled list to the author of the inspected item.

The inspector reviews the product for general problems as well as those related to their specific area of expertise. Checklists are used during this stage for guidance on typical types of defects that are found in the type of product being inspected. The product being inspected is also checked against standard documents to assure compliance and correctness. After reviewing, the inspectors record the defects found on a log and the time spent during preparation. Completed preparation logs are submitted to the moderator prior to the inspection meeting.

The moderator reviews the logs submitted by each inspector to determine whether the team is adequately prepared. The moderator also checks for trouble spots that may need extra attention during inspection, common defects that can be categorized quickly, and the areas of major concern. If the logs indicate that the team is not adequately prepared, the moderator should reschedule the inspection meeting. After this, the compiled log file is submitted to the author.

Inspection meeting Once all the initial preparation is complete, the actual inspection meeting can start. The inspection meeting starts with the author of the inspected item who has created it. The author first discusses every issue raised by different members in the compiled log file. After the discussion, all the members arrive at a consensus whether the issues pointed out are in fact errors and if they are errors, should they be admitted by the author. It may be possible that during the discussion on any issue, another error is found. Then, this new error is also discussed and recorded as an error by the author.

The basic goal of the inspection meeting is to uncover any bug in the item. However, no effort is made in the meeting to fix the bug. It means that bugs are only being notified to the author, which he will fix later. If there is any clarification regarding these bugs, then it should be asked or discussed with other members during the meeting.

Another fact regarding the inspection is that the members of the meeting should be sensitive to the feelings of the author. The author should not be attacked by other members such that the author feels guilty about the product. Every activity in the meeting should be a constructive engagement so that more and more bugs can be discovered. It is the duty of the moderator that the meeting remains focused towards its objective and the author is not discouraged in any way.

At the end, the moderator concludes the meeting and produces a summary of the inspection meeting. This summary is basically a list of errors found in the item that need to be resolved by the author.

Rework The summary list of the bugs that arise during the inspection meeting needs to be reworked by the author. The author fixes all these bugs and reports back to the moderator.

Follow-up It is the responsibility of the moderator to check that all the bugs found in the last meeting have been addressed and fixed. He prepares a report and ascertains that all issues have been resolved. The document is then approved for release. If this is not the case, then the unresolved issues are mentioned in a report and another inspection meeting is called by the moderator.

6.1.3 Benefits of Inspection Process

Bug reduction The number of bugs is reduced through the inspection process. L.H. Fenton [86] reported that through the inspection process in IBM, the number of bugs per thousand lines of code has been reduced by two-thirds. Thus, inspection helps reduce bug injection and detection rates. According to Capers Jones, 'Inspection is by far the most effective way to remove bugs.'

Bug prevention Inspections can also be used for bug prevention. Based on the experiences of previous inspections, analysis can be made for future inspections or projects, thereby preventing the bugs which have appeared earlier. Programmers must understand why bugs appear and what can be done to avoid them in future. At the same time, they should provide inspection results to the quality assurance team.

Productivity Since all phases of SDLC may be inspected without waiting for code development and its execution, the cost of finding bugs decreases, resulting in an increase in productivity. Moreover, the errors are found at their exact places, therefore reducing the need of dynamic testing and debugging. In the article by Fagan [43], an increase of 23% in coding productivity and a 25% reduction in schedules were reported.

Real-time feedback to software engineers The inspections also benefit software engineers/developers because they get feedback on their products on a relatively real-time basis. Developers find out the type of mistakes they make and what is the error density. Since they get this feedback in the early stages of development, they may improve their capability. Thus, inspections benefit software engineers/developers in the sense that they can recognize their weakness and improve accordingly, which in turn benefits the cost of the project.

Reduction in development resource The cost of rework is surprisingly high if inspections are not used and errors are found during development or testing. Therefore, techniques should be adopted such that errors are found and fixed as close to their place of origin as possible. Inspections reduce the effort required for dynamic testing and any rework during design and code, thereby causing an overall net reduction in the development resource. Without inspections, more resources may be required during design and dynamic testing. But, with inspection, the resource requirement is greatly reduced.

Quality improvement We know that the direct consequence of testing is improvement in the quality of software. The direct consequence of static testing also results in the improvement of quality of the final product. Inspections help to improve the quality by checking the standard compliance, modularity, clarity, and simplicity.

Project management A project needs monitoring and control. It depends on some data obtained from the development team. However, this data cannot be relied on forever. Inspection is another effective tool for monitoring the progress of the project.

Checking coupling and cohesion The modules' coupling and cohesion can be checked easily through inspection as compared to dynamic testing. This also reduces the maintenance work.

Learning through inspection Inspection also improves the capability of different team members, as they learn from the discussions on various types of bugs and the reasons why they occur. It may be more beneficial for new members. They can learn about the project in a very short time. This helps them in the later stages of development and testing.

Process improvement There is always scope of learning from the results of one process. An analysis of why the errors occurred or the frequent places where the errors occurred can be done by the inspection team members. The analysis results can then be used to improve the inspection process so that the current as well as future projects can benefit. Discussed below are the issues of process improvement.

Finding most error-prone modules Through the inspection process, the modules can be analysed based on the error-density of the individual module, as shown in Table 6.1.

Table 6.1 Error-prone modules

Module Name	Error density (Error/ KLoC)
A	23
B	13
C	45

From the example modules given in Table 6.1, Module C is more error-prone. This information can be used and some decision should be taken as to:

(i) Redesign the module

(ii) Check and rework on the code of Module C

(iii) Take extra precautions and efficient test cases to test the module

Distribution of error-types Inspections can also be used to provide data according to the error-types. It means that we can analyse the data of the percentage of bugs in a particular type of bug category, as shown in Table 6.2.

Table 6.2 Bug type distribution

Bug Type	No. of errors	%
Design Bugs	78	57.8
Interface Bugs	45	33.4
Code Bugs	12	8.8

If we get this data very early in the development process, then we can analyse which particular type of bugs are repeating. We can modify the process well in time and the results can also be used in future.

In this way, as the organization gains experience, the process can be improved.

6.1.4 Effectiveness of Inspection Process

In an analysis done by Oliver Laitenberger [107], the inspection process was found to be effective as compared to structural testing because the inspection process alone found 52% errors. Of the remaining 48% of defects, on an average, only 17% are detected by structural testing.

The effectiveness of the inspection process lies in its rate of inspection. The rate of inspection refers to how much evaluation of an item has been done by the team. If the rate is very fast, it means the coverage of item to be evaluated is high. Similarly if the rate is too slow, it means that the coverage is not much. However, the rate should also be considered in the perspective of detection of errors. If the rate is too fast, then it may be possible that it reveals only few errors. If the rate is too slow, then the cost of project increases. Thus, a balanced rate of inspection should be followed so that it concentrates on a reasonable number of defects. It may be calculated as the error detection efficiency of the inspection process, as given below:

$$\text{Error detection efficiency} = \frac{\text{Error found by an inspection}}{\text{Total errors in the item before inspection}} \times 100$$

The rate of inspection also depends on some factors like the experience of the inspection team, the programming language, and the application domain.

6.1.5 Cost of Inspection Process

With at least four members involved in an inspection team, the cost of inspecting 100 lines of code is roughly equivalent to one person-day of effort. This

assumes that the inspection itself takes about an hour and that each member spends 1-2 hours preparing for the inspection. Testing costs are variable and depend on the number of faults in the software. However, the effort required for the program inspection is less than half the effort that would be required for equivalent dynamic testing. Further, it has been estimated that the cost of inspection can be 5–10% of the total cost of the project.

6.1.6 Variants of Inspection process

After Fagan's original formal inspection concept, many researchers proposed modifications in it. Table 6.3 lists some of the variants of the formal inspection.

Table 6.3 Formal inspection variants

Active Design Reviews (ADRs)	Several reviews are conducted targeting a particular type of bugs and conducted by the reviewers who are experts in that area.
Formal Technical Asynchronous review method (FTArm)	Inspection process is carried out without really having a meeting of the members. This is a type of asynchronous inspection in which the inspectors never have to simultaneously meet.
Gilb Inspection	Defect detection is carried out by individual inspector at his level rather than in a group.
Humphrey's Inspection Process	Preparation phase emphasizes the finding and logging of bugs, unlike Fagan inspections. It also includes an analysis phase wherein individual logs are analysed and combined into a single list.
N-Fold inspections	Inspection process's effectiveness can be increased by replicating it by having multiple inspection teams.
Phased Inspection	Phased inspections are designed to verify the product in a particular domain by experts in that domain only.
Structured Walkthrough	Described by Yourdon. Less formal and rigorous than formal inspections. Roles are coordinator, scribe, presenter, reviewers, maintenance oracle, standards bearer, user representative. Process steps are Organization, Preparation, Walkthrough, and Rework. Lacks data collection requirements of formal inspections.

Active Design Reviews

Active Design Reviews [87] are for inspecting the design stage of SDLC. In this process, several reviews are conducted targeting a particular type of bugs and conducted by the reviewers who are experts in that area. Thus, it covers all the sections of the design document based on several small reviews instead of only one inspection. It is also based on the use of questionnaires to give the reviewers better-defined responsibilities and to make them play a more active role. The questionnaires are designed to make the reviewers take an active

stand on issues and to use the documentation. The technique has been used in the Naval Research Laboratory's (Washington) software cost reduction (SCR) project for several years with good results. The SCR project involves the experimental redevelopment of the operational flight program (OFP) for the Navy's A-7E aircraft.

These are conducted in the following stages (see Fig. 6.2):

Figure 6.2 Active design reviews process

Overview A brief overview of the module being reviewed is presented. The overview explains the modular structure in case it is unfamiliar to reviewers, and shows them where the module belongs in the structure.

Review Reviewers are assigned sections of the document to be reviewed and questionnaires based on the bug type. Reviewers answer the questions in the questionnaires. They are also assigned a timeframe during which they may raise any questions they have, and a time to meet with designers after the designers have read the completed questionnaires.

Meeting The designers read the completed questionnaires and meet the reviewers to resolve any queries that the designers may have about the reviewer's answers to the questionnaires. The reviewers may be asked to defend their answers. This interaction continues until both designers and reviewers understand the issues, although an agreement on these issues may not be reached. After the review, the designers produce a new version of the documentation.

Formal Technical Asynchronous Review Method (FTArm)

In this process, the meeting phase of inspection is considered expensive and therefore, the idea is to eliminate this phase. The inspection process is carried out without having a meeting of the members. This is a type of asynchronous inspection [88] in which the inspectors never have to simultaneously meet. For this process, an online version of the document is made available to every member where they can add their comments and point out the bugs. This process consists of the following steps, as shown in Fig. 6.3.

Setup It involves choosing the members and preparing the document for an asynchronous inspection process. The document is prepared as a hypertext document.

Figure 6.3 Asynchronous method

Orientation It is same as the overview step of the inspection process discussed earlier.

Private review It is same as the preparation phase discussed in the inspection process. Here, each reviewer or inspector individually gives his comments on the document being inspected. However, each inspector provides comments individually and is unable to see the other inspector's comments.

Public review In this step, all comments provided privately are made public and all inspectors are able to see each other's comments and put forward their suggestions. Based on this feedback and with consultation of the author, it is decided that there is a bug.

Consolidation In this step, the moderator analyses the result of private and public reviews and lists the findings and unresolved issues, if any.

Group meeting If required, any unresolved issues are discussed in this step. But the decision to conduct a group meeting is taken in the previous step only by the moderator.

Conclusion The final report of the inspection process along with the analysis is produced by the moderator.

Gilb Inspection

Gilb and Graham [89] defined this process. It differs from Fagan inspection in that the defect detection is carried out by individual inspectors at their own level rather than in a group. Therefore, a checking phase has been introduced. Three different roles are defined in this type of inspection:

- **Leader** is responsible for planning and running the inspection.
- **Author of the document**
- **Checker** is responsible for finding and reporting the defects in the document.

The inspection process consists of the following steps, as shown in Fig. 6.4.

Figure 6.4 Gilb inspection process

Entry The document must pass through an entry criteria so that the inspection time is not wasted on a document which is fundamentally flawed.

Planning The leader determines the inspection participants and schedules the meeting.

Kick-off The relevant documents are distributed, participants are assigned roles and briefed about the agenda of the meeting.

Checking Each checker works individually and finds defects.

Logging Potential defects are collected and logged.

Brainstorm In this stage, process improvement suggestions are recorded based on the reported bugs.

Edit After all the defects have been reported, the author takes the list and works accordingly.

Follow-up The leader ensures that the edit phase has been executed properly.

Exit The inspection must pass the exit criteria as fixed for the completion of the inspection process.

Humphrey's Inspection Process

It was described by Watts Humphrey [90]. In this process, the preparation phase emphasizes on finding and logging of bugs, unlike Fagan inspections. It also includes an analysis phase between preparation and meeting. In the analysis phase, individual logs are analysed and combined into a single list. The steps of this process are shown in Fig. 6.5.

Figure 6.5 Humphrey's process

N-Fold Inspection

It is based on the idea that the effectiveness of the inspection process can be increased by replicating it [91]. If we increase the number of teams inspecting the item, the percentage of defects found may increase. But sometimes the cost of organizing multiple teams is higher as compared to the number of defects found. A proper evaluation of the situation is required. Originally, this process was used for inspecting requirement specifications, but it can also be used for any phase.

As discussed, this process consists of many independent inspection teams. This process needs a coordinator who coordinates various teams, collects and collates the inspection data received by them. For this purpose, he also meets the moderator of every inspection team. This process consists of the following stages, as shown in Fig. 6.6.

Planning and overview This is the formal planning and overview stage. In addition, it also includes the planning of how many teams will participate in the inspection process.

Inspection stages There are many inspection processes adopted by many teams. It is not necessary that every team will choose the same inspection process. The team is free to adopt any process.

Collation phase The results from each inspection process are gathered, collated, and a master list of all detected defects is prepared.

Rework and follow-up This step is same as the tradition Fagan inspection process.

Figure 6.6 N-fold inspection

Phased Inspection

Phased inspections [92] are designed to verify the product in a particular domain. If we feel that in a particular area, we may encounter errors, then phased inspections are carried out. But for this purpose, only experts who have experience in that particular domain are called. Thus, this inspection process provides a chance to utilize human resources. In this process, inspection is divided into more than one phase. There is an ordered set of phases

and each phase is designed such that it will check a particular feature in the product. The next phase in the ordered set of phases assumes that the particular feature has been verified in the previous phase. There are two types of phases, as discussed below.

Single inspector In this phase, a rigorous checklist is used by a single inspector to verify whether the features specified are there in the item to be inspected.

Multiple inspector Here, a checklist cannot be used. It means that the item cannot be verified just with the questions mentioned in the checklist. There are many inspectors who are distributed the required documents for verification of an item. Each inspector examines this information and develops a list of questions of their own based on a particular feature. These questions are not in binary form as in single inspection. The item is then inspected individually by all the inspectors based on a self-developed checklist which is either application or domain specific. After individual checking by the inspectors, a reconciliation meeting is organized where inspectors compare their findings about the item.

Structured Walkthrough

Though structured walkthrough is a variant of the inspection process, we have now categorized it as a separate category for static testing. We will discuss structured walkthrough in Section 6.2.

6.1.7 Reading Techniques

A reading technique can be defined as a series of steps or procedures whose purpose is to guide an inspector to acquire a deep understanding of the inspected software product. Thus, a reading technique can be regarded as a mechanism or strategy for the individual inspector to detect defects in the inspected product. Most of the techniques found in the literature support individual inspection work. The various reading techniques are discussed below.

Ad hoc method In this method, there is no direction or guidelines provided for inspection. However, ad hoc does not mean that inspection participants do not scrutinize the inspected product systematically. The word *ad hoc* only refers to the fact that no technical support on how to detect defects in a software artifact is given to them. In this case, defect detection fully depends on the skills, knowledge, and experience of an inspector.

Checklists A checklist is a list of items that focus the inspector's attention on specific topics, such as common defects or organizational rules, while reviewing a software document [43]. The purpose of checklists is to gather expertise

concerning the most common defects and thereby supporting inspections. The checklists are in the form of questions which are very general and are used by the inspection team members to verify the item.

However, checklists have some drawbacks too:

- The questions, being general in nature, are not sufficiently tailored to take into account a particular development environment.
- Instruction about using a checklist is often missing.
- There is a probability that some defects are not taken care of. It happens in the case of those type of defects which have not been previously detected.

Scenario-based reading Checklists are general in nature and do not cover different types of bugs. Scenario-based reading [93] is another reading technique which stresses on finding different kind of defects. It is based on scenarios, wherein inspectors are provided with more specific instructions than typical checklists. Additionally, they are provided with different scenarios, focusing on different kind of defects.

Basili *et al.* [94] define scenario-based reading as a high-level term, which they break down to more specific techniques. The original method of Porter & Votta [93] is also known as *defect-based reading.* By using the definition by Basili *et al.,* most of the reading techniques are based on scenario-based techniques wherein the inspector has to actively work with the inspected documents instead of mere straightforward reading. The following methods have been developed based on the criteria given by scenario-based reading.

Perspective-based reading The idea behind this method is that a software item should be inspected from the perspective of different stakeholders [95,104,105]. Inspectors of an inspection team have to check the software quality as well as the software quality factors of a software artifact from different perspectives. The perspectives mainly depend upon the roles people have within the software development or maintenance process. For each perspective, either one or multiple scenarios are defined, consisting of repeatable activities an inspector has to perform, and the questions an inspector has to answer. For example, a testing expert first creates a test plan based on the requirements specification and then attempts to find defects from it.

This is designed specifically for requirements inspection, and inspectors who participate in inspection should have expertise in inspecting requirements of their own area of expertise. Later, perspective-based reading has been applied to code inspections and design inspections.

Usage-based reading This method proposed by Thelin *et al.* [96,97,98] is applied in design inspections. Design documentation is inspected based on use

cases, which are documented in requirements specification. Since use-cases are the basis of inspection, the focus is on finding functional defects, which are relevant from the users' point of view.

Abstraction driven reading This method given by Dunsmore *et al.* [99,100,101] is designed for code inspections. In this method, an inspector reads a sequence of statements in the code and abstracts the functions these statements compute. An inspector repeats this procedure until the final function of the inspected code artifact has been abstracted and can be compared to the specification. Thus, the inspector creates an abstraction level specification based on the code under inspection to ensure that the inspector has really understood the code.

Task-driven reading This method is also valid for code inspections proposed by Kelly & Shepard [102]. In this method, the inspector has to create a data dictionary, a complete description of the logic and a cross-reference between the code and the specifications.

Function-point based scenarios This is based on scenarios for defect detection in requirements documents [103]. This approach is based on function-point analysis (FPA). FPA defines a software system in terms of its inputs, files, inquiries, and outputs. The scenarios designed around function-points are known as *function-point scenarios.* It consists of questions and directs the focus of an inspector to a specific function-point item within the inspected requirements document.

6.1.8 Checklists for Inspection Process

The inspection team must have a checklist against which they detect errors. The checklist is according to the item to be inspected. For example, the design document and the code of the module should have different checklists. Checklists can be prepared with the points mentioned in the verification of each phase. Checklists should be prepared in consultation with experienced staff and regularly updated as more experience is gained by the inspection process.

The checklist may vary according to the environment and needs of the organization. Each organization should prepare its own checklists for every item.

6.2 Structured Walkthroughs

The idea of *structured walkthroughs* was proposed by Yourdon [106]. It is a less formal and less rigorous technique as compared to inspection. The common term used for static testing is *inspection* but it is a very formal process. If you

want to go for a less formal process having no bars of organized meeting, then walkthroughs are a good option.

A typical structured walkthrough team consists of the following members:

- **Coordinator** Organizes, moderates, and follows up the walkthrough activities.
- **Presenter/Developer** Introduces the item to be inspected. This member is optional.
- **Scribe/Recorder** Notes down the defects found and suggestion proposed by the members.
- **Reviewer/Tester** Finds the defects in the item.
- **Maintenance Oracle** Focuses on long-term implications and future maintenance of the project.
- **Standards Bearer** Assesses adherence to standards.
- **User Representative/Accreditation Agent** Reflects the needs and concerns of the user.

Walkthroughs differ significantly from inspections. An inspection is a six-step, rigorous, formalized process. The inspection team uses the checklist approach for uncovering errors. A walkthrough is less formal, has fewer steps and does not use a checklist to guide or a written report to document the team's work. Rather than simply reading the program or using error checklists, the participants 'play computer'. The person designated as a tester comes to the meeting armed with a small set of paper test cases—representative sets of inputs and expected outputs for the program or module. During the meeting, each test case is mentally executed. That is, the test data are walked through the logic of the program. The state of the program is monitored on a paper or any other presentation media. The walkthrough should have a follow-up process similar to that described in the inspection process. The steps of a walkthrough process are shown in Fig. 6.7.

Figure 6.7 Walkthrough process

6.3 Technical Reviews

A technical review is intended to evaluate the software in the light of development standards, guidelines, and specifications and to provide the management with evidence that the development process is being carried out according to

the stated objectives. A review is similar to an inspection or walkthrough, except that the review team also includes management. Therefore, it is considered a higher-level technique as compared to inspection or walkthrough.

A technical review team is generally comprised of management-level representatives and project management. Review agendas should focus less on technical issues and more on oversight than an inspection. The purpose is to evaluate the system relative to specifications and standards, recording defects and deficiencies. The moderator should gather and distribute the documentation to all team members for examination before the review. He should also prepare a set of indicators to measure the following points:

- Appropriateness of the problem definition and requirements
- Adequacy of all underlying assumptions
- Adherence to standards
- Consistency
- Completeness
- Documentation

The moderator may also prepare a checklist to help the team focus on the key points. The result of the review should be a document recording the events of the meeting, deficiencies identified, and review team recommendations. Appropriate actions should then be taken to correct any deficiencies and address all recommendations.

SUMMARY

Static testing techniques are not as popular as dynamic testing techniques. However, researchers have found static testing to be an effective aid in finding and preventing the defects at an early stage. In recent years, due to the importance of early testing, these techniques are finding their place in industries.

This chapter introduces static testing as a complementary technique to dynamic testing. Static testing reveals the bugs which cannot be detected by dynamic testing. We have taken three categories of static testing, namely inspection, walkthroughs, and reviews, and distinguished them. Inspection process is a widely used term for static testing. A lot of terminologies are used interchangeably for static testing.

We have discussed every aspect of inspection process including the team members and the inspection process. Moreover, in the literature, many variants of this process are also available. These variants have also been discussed.

Let us review the important concepts discussed in this chapter:

- The inspection process is a formal in-process manual examination of an item to detect bugs. It may be applied to any product or partial product of the software development process. This process is carried out by a group of peers with the help of checklists.

- The rate of inspection is calculated as the error detection efficiency of the inspection process as given below:

$$\text{Error detection efficiency} = \frac{\text{Error found by an inspection}}{\text{Total errors in the item before inspection}} \times 100$$

- Active Design Reviews inspects the design stage of SDLC. In this process, several reviews are conducted targeting a particular type of bugs.
- Formal Technical Asynchronous review method (FTArm) is a type of asynchronous inspection in which the inspectors never have to simultaneously meet. For this process, an online version of the document is made available to every member where they can add their comments and point out the bugs.
- Gilb Inspection is a type of inspection process wherein the defect detection is carried out by individual inspectors at their own level rather than in a group.
- Humphrey's Inspection emphasizes finding and logging the bugs in the preparation phase. It also includes an analysis phase between preparation and meeting. In the analysis phase, individual logs are analysed and combined into a single list.
- N-Fold Inspection process consists of many independent inspection teams to increase the effectiveness of the inspection process and results in detecting more number of bugs quickly. This process needs a coordinator who coordinates various teams, collects and collates the inspection data received by them.
- Phased Inspections are designed to verify the product in a particular domain with the help of experts in that domain. In this process, the inspection is divided into more than one phase. There is an ordered set of phases and each phase is designed such that it will check a particular feature in the product.
- A Reading Technique can be regarded as a mechanism or strategy for the individual inspector to detect defects in the inspected product.
- A Walkthrough is less formal, has fewer steps, and does not use a checklist to guide or a written report to document the team's work.
- A Technical Review is intended to evaluate the software in light of development standards, guidelines, and specifications and to provide the management with evidence that the development process is being carried out according to the stated objectives. A review is similar to an inspection or walkthrough, except that the review team also includes management.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. In static testing, a bug is found at its _____ location.
   (a) Exact
   (b) Nearby
   (c) None of the above
2. Static testing can be applied for most of the _____ .
   (a) Validation activities

(b) Verification activities
(c) SDLC activities
(d) None of the above

3. Formal peer evaluation of a software element whose objective is to verify that the software element satisfies its specifications and conforms to standards, is called _____ .
   (a) Walkthrough
   (b) Inspections
   (c) Reviews
   (d) None of the above
4. The programmer or designer responsible for producing the program or document is known as _____ .
   (a) Author
   (b) Owner
   (c) Producer
   (d) All
5. The person who finds errors, omissions, and inconsistencies in programs and documents during an inspection is known as _____ .
   (a) Inspector
   (b) Moderator
   (c) Author
   (d) Producer
6. The key person with the responsibility of planning and successful execution of inspection is known as _____ .
   (a) Inspector
   (b) Moderator
   (c) Author
   (d) Producer
7. The inspection team points out any potential errors or problems found and records them in _____.
   (a) SDD
   (b) SRS
   (c) STD
   (d) Log Form
8. 'How much evaluation of an item has been done by the team' is called _____ .
   (a) Rate of errors
   (b) Rate of inspection
   (c) Rate of failures
   (d) None of the above
9. _____ is a more formal process.
   (a) Walkthroughs
   (b) Inspection

(c) Reviews
(d) None of the above

10. The efficiency of code coverage performed by dynamic testing _____ with the increase in size of the system.
    (a) Decreases
    (b) Increases
    (c) Remains same
    (d) None of the above
11. Through the inspection process, the modules can be analysed based on _____.
    (a) Error-types
    (b) Inspection reports
    (c) Error-density
    (d) None of the above
12. The inspection in which the inspectors never have to simultaneously meet is known as _____.
    (a) Phased Inspection
    (b) FTArm
    (c) Gilb Inspection
    (d) All
13. Checking phase has been introduced in _____.
    (a) Phased Inspection
    (b) FTArm
    (c) Gilb Inspection
    (d) None of the above
14. Analysis phase between preparation and meeting has been introduced in _____.
    (a) Phased Inspection
    (b) Humphrey's Inspection
    (c) FTArm
    (d) N-fold Inspection
15. Collation Phase has been introduced in _____.
    (a) Phased Inspection
    (b) Humphrey's Inspection
    (c) FTArm
    (d) N-fold Inspection
16. _____ process gives the chance to utilize human resources.
    (a) Phased Inspection
    (b) Humphrey's Inspection
    (c) FTArm
    (d) N-fold Inspection
17. A series of steps or procedures whose purpose is to guide an inspector in acquiring a deep understanding of the inspected software is known as _____.

(a) Checklists
(b) Inspection
(c) Reading Techniques
(d) N-fold Inspection

18. _____ is a reading technique.
    (a) Checklists
    (b) Inspection
    (c) Usage-based method
    (d) Task-based method
19. A review is similar to an inspection or walkthrough, except that the review team also includes _____.
    (a) Customer
    (b) Developer
    (c) Tester
    (d) Management
20. _____ is not an inspection variant.
    (a) Active design review
    (b) FTArm
    (c) Walkthrough
    (d) None of the above

Review Questions

1. What are the advantages of static testing as compared to dynamic testing?
2. What are the benefits of inspection process as compared to dynamic testing?
3. Who can be a member of the inspection team?
4. What are the stages of an inspection process?
5. How does the rate of inspection affect the effectiveness of the inspection process?
6. What is the difference between inspection, walkthrough, and reviews?
7. Make a table indicating the feature of every type of inspection variant.
8. What are the factors that increase the effectiveness of N-Fold inspection?
9. How is scenario-based reading different from checklists? Explain the types of scenario-based reading.
10. Take a small project and apply all the static testing techniques on its SRS, SDD, and code.
11. Develop a list based on usage-based reading method on the project taken in Problem 10.
12. What are the drawbacks of checklists?
13. Is ad-hoc method a random method of reading?

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

7

Validation Activities

OBJECTIVES

After reading this chapter, you should be able to understand:

- Validation is the next step after verification
- Validation is performed largely by black-box testing techniques
- Unit validation testing and role of stubs and drivers in unit validation
- Integration testing and its types: Decomposition-based integration, call graph-based integration, path graph-based integration testing
- Types of Decomposition-based integration: Top-down integration, Bottom-up integration testing
- Function testing
- System testing and its types: Recovery testing, Security testing, Stress testing, Performance testing, Usability testing, Compatibility testing
- Acceptance testing and its types: Alpha and Beta testing

In this chapter, we will discuss the details of all validation concepts. Once we have verified every activity/stage in SDLC as shown in Fig. 7.1, the software or module built up till this stage should also be validated. By validation, we mean that the module that has been prepared till now is in conformance with the requirements which were set initially in the SRS or user manual.

As shown in Fig. 7.1, every validation testing focuses on a particular SDLC phase and thereby focuses on a particular class of errors. We can see a one-to-one correspondence between development and testing processes. For example, the purpose of unit validation testing is to find discrepancies between the developed module's functionality and its requirements and interfaces specified in the SRS. Similarly, the purpose of system validation testing is to explore whether the product is consistent with its original objectives. The advantage of this structure of validation testing is that it avoids redundant testing and prevents one from overlooking large classes of errors.

Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements [7]. A test plan outlines the classes of tests to be conducted and a test procedure defines specific test cases that will be used to demonstrate conformity with requirements. Both the plan and procedure are designed to ensure that all functional requirements are satisfied, all behavioural character-

istics are achieved, all performance requirements are attained, documentation is correct and human-engineered, and other requirements are met.

Figure 7.1 V&V activities

Validation testing techniques are described in the following sections.

7.1 Unit Validation Testing

Since unit is the smallest building block of the software system, it is the first piece of system to be validated. Before we validate the entire software, units or modules must be validated. Unit testing is normally considered an adjunct to the coding step. However, it has been discussed that testing with the code of a unit is called the *verification step.* Units must also be validated to ensure that every unit of software has been built in the right manner in conformance with user requirements. Unit tests ensure that the software meets at least a baseline level of functionality prior to integration and system testing. While

developing the software, if the developer detects and removes the bug, it offers significant savings of time and costs. This type of testing is largely based on black-box techniques.

Though software is divided into modules but a module is not an isolated entity. The module under consideration might be getting some inputs from another module or the module is calling some other module. It means that a module is not independent and cannot be tested in isolation. While testing the module, all its interfaces must be simulated if the interfaced modules are not ready at the time of testing the module under consideration. Discussed below are the types of interface modules which must be simulated, if required, to test a module.

Drivers Suppose a module is to be tested, wherein some inputs are to be received from another module. However, this module which passes inputs to the module to be tested is not ready and under development. In such a situation, we need to simulate the inputs required in the module to be tested. For this purpose, a main program is prepared, wherein the required inputs are either hard-coded or entered by the user and passed on to the module under test. This module where the required inputs for the module under test are simulated for the purpose of module or unit testing is known as a *driver module.* The driver module may print or interpret the results produced by the module under testing.

For example, see the design hierarchy of the modules, as shown in Fig. 7.2. Suppose module B is under test. In the hierarchy, module A is a superordinate of module B. Suppose module A is not ready and B has to be unit tested. In this case, module B needs inputs from module A. Therefore, a driver module is needed which will simulate module A in the sense that it passes the required inputs to module B and acts as a main program for module B in which its being called, as shown in Fig. 7.3.

Figure 7.2 Design hierarchy of an example system

Figure 7.3 Driver Module for module A

Therefore, it can be said that a test driver is supporting the code and data used to provide an environment for testing a part of a system in isolation. In fact, it drives the unit being tested by creating necessary 'inputs' required for the unit and then invokes the unit. A test driver may take inputs in the following form and call the unit to be tested:

- It may hard-code the inputs as parameters of the calling unit.
- It may take the inputs from the user.
- It may read the inputs from a file.

Thus, a driver can be defined as a software module which is used to invoke a module under test and provide test inputs, control and monitor execution, and report test results or most simplistically a line of code that calls a method and passes a value to that method.

A test driver provides the following facilities to a unit to be tested:

- Initializes the environment desired for testing.
- Provides simulated inputs in the required format to the units to be tested.

Projects where commercial drivers are not available, specialized drivers need to be developed. This happens mainly in defence projects where projects are developed for a special application.

Stubs The module under testing may also call some other module which is not ready at the time of testing. Therefore, these modules need to be simulated for testing. In most cases, dummy modules instead of actual modules, which are not ready, are prepared for these subordinate modules. These dummy modules are called *stubs*.

Thus, a stub can be defined as a piece of software that works similar to a unit which is referenced by the unit being tested, but it is much simpler than the actual unit. A stub works as a 'stand-in' for the subordinate unit and provides the minimum required behaviour for that unit.

For example, consider Fig. 7.2 again. Module B under test needs to call module D and module E. But they are not ready. So there must be some skeletal structure in their place so that they act as dummy modules in place of the actual modules. Therefore, stubs are designed for module D and module E, as shown in Fig. 7.4.

Stubs have the following characteristics:

- Stub is a place holder for the actual module to be called. Therefore, it is not designed with the functionalities performed by the actual module. It is a reduced implementation of the actual module.
- It does not perform any action of its own and returns to the calling unit (which is being tested).
- We may include a display instruction as a trace message in the body of stub. The idea is that the module to be called is working fine by accepting the input parameters.
- A constant or null must be returned from the body of stub to the calling module.
- Stub may simulate exceptions or abnormal conditions, if required.

Figure 7.4 Stubs

Benefits of Designing Stubs and Drivers

The benefit of designing drivers and stubs (see Fig. 7.5) is that a unit will work/behave in the simulated environment as in the actual software environment. Now a unit test case can be written against the interface and the unit will still work properly once the drivers and stubs have been placed correctly.

The benefits of designing stubs and drivers are:

- Stubs allow the programmer to call a method in the code being developed, even if the method does not have the desired behaviour yet.
- By using stubs and drivers effectively, we can cut down our total debugging and testing time by testing small parts of a program individually, helping us to narrow down problems before they expand.

- Stubs and drivers can also be an effective tool for demonstrating progress in a business environment. For example, if you are to implement four specific methods in a class by the end of the week, you can insert stubs for any method and write a short driver program so that you can demonstrate to your manager or client that the requirement has been met. As you continue, you can replace the stubs and drivers with the real code to finish your program.

Figure 7.5 Drivers and Stubs

However, drivers and stubs represent overheads also. Overhead of designing them may increase the time and cost of the entire software system. Therefore, they must be designed simple to keep overheads low. Stubs and drivers are generally prepared by the developer of the module under testing. Developers use them at the time of unit verification. But they can also be used by any other person who is validating the unit.

Example 7.1

Consider the following program:

```
main()
{
        int a,b,c,sum,diff,mul;

        scanf("%d %d %d", &a, &b, &c);
        sum = calsum(a,b,c);
        diff = caldiff(a,b,c);
        mul = calmul(a,b,c);
        printf("%d %d %d", sum, diff, mul);
}

calsum(int x, int y, int z)
{
        int d;
```

```
        d = x + y + z;
        return(d);
}
```

(a) Suppose `main()` module is not ready for the testing of `calsum()` module. Design a driver module for `main()`.

(b) Modules `caldiff()` and `calmul()` are not ready when called in `main()`. Design stubs for these two modules.

Solution

(a) **Driver for `main()` module:**

```
main()
  {
        int a, b, c, sum;

        scanf("%d %d %d", &a, &b, &c);
        sum = calsum(a,b,c);
        printf("The output from calsum module is %d", sum);
}
```

(b) **Stub for `caldiff()` Module**

```
caldiff(int x, int y, int z)
{
        printf("Difference calculating module");
        return 0;
}
```

Stub for `calmul()` Module

```
calmul(int x, int y, int z)
{
        printf("Multiplication calculation module");
        return 0;
}
```

7.2 Integration Testing

In the modular design of a software system where the system is composed of different modules, integration is the activity of combining the modules together when all the modules have been prepared. Integration of modules is according to the design of software specified earlier. Integration aims at constructing a working software system. But a working software demands full testing and thus, integration testing comes into the picture.

Why do we need integration testing? When all modules have been verified independently, then why is integration testing necessary? As discussed earlier, modules are not standalone entities. They are a part of a software system which comprises of many interfaces. Even if a single interface is mismatched, many modules may be affected. Thus, integration testing is necessary for the following reasons:

- Integration testing exposes inconsistency between the modules such as improper call or return sequences.
- Data can be lost across an interface.
- One module when combined with another module may not give the desired result.
- Data types and their valid ranges may mismatch between the modules.

Thus, integration testing focuses on bugs caused by interfacing between the modules while integrating them.

There are three approaches for integration testing, as shown in Fig. 7.6.

Figure 7.6 Integration Methods

7.2.1 Decomposition-Based Integration

The idea for this type of integration is based on the decomposition of design into functional components or modules. The functional decomposition is shown as a tree in Fig. 7.7. In the tree designed for decomposition-based integration, the nodes represent the modules present in the system and the links/edges between the two modules represent the calling sequence. The nodes on the last level in the tree are *leaf nodes*.

In the tree structure shown in Fig. 7.7, module A is linked to three subordinate modules, B, C, and D. It means that module A calls modules, B, C, and D. All integration testing methods in the decomposition-based integration assume that all the modules have been unit tested in isolation. Thus, with the decomposition-based integration, we want to test the interfaces among separately tested modules.

Figure 7.7 Decomposition Tree

Integration methods in decomposition-based integration depend on the methods on which the activity of integration is based. One method of integrating is to integrate all the modules together and then test it. Another method is to integrate the modules one by one and test them incrementally. Based on these methods, integration testing methods are classified into two categories: (a) non-incremental and (b) incremental.

Non-Incremental Integration Testing

In this type of testing, either all untested modules are combined together and then tested or unit tested modules are combined together. It is also known as Big-Bang integration testing.

Big-Bang method cannot be adopted practically. This theory has been discarded due to the following reasons:

1. Big-Bang requires more work. For example, consider the following hierarchy of a software system.

Figure 7.8 Integration Testing

According to the Big-Bang theory, if all unit tested modules are integrated in this example, then for unit testing of all the modules independently, we require four drivers and seven stubs. This count will grow according to the size of the system.

2. Actual modules are not interfaced directly until the end of the software system.
3. It will be difficult to localize the errors since the exact location of bugs cannot be found easily.

Incremental Integration Testing

In this type, you start with one module and unit test it. Then combine the module which has to be merged with it and perform test on both the modules. In this way, incrementally keep on adding the modules and test the recent environment. Thus, an integrated tested software system is achieved.

Incremental integration testing is beneficial for the following reasons:

1. Incremental approach does not require many drivers and stubs.
2. Interfacing errors are uncovered earlier.
3. It is easy to localize the errors since modules are combined one by one. The first suspect is the recently added module. Thus, debugging becomes easy.
4. Incremental testing is a more thorough testing. Consider the example in Fig. 7.8. If we are testing module 6, then although module 1 and 2 have been tested previously, they will get the chance to be tested again and again. This gives the modules more exposure to the probable bugs by the time the last module in the system is tested.

However, the two integration methods discussed above are sometimes not acceptable. Incremental testing suffers from the problem of serially combining the methods according to the design. But practically, sometimes it is not feasible in the sense that all modules are not ready at the same time. Therefore, incremental testing cannot be adopted in its pure form.

One method that may work is to borrow the good features of both the approaches. According to the big-bang method, all modules should be unit-tested independently as they are developed. In this way, there is parallelism. As soon as one module is ready, it can be combined and tested again in the integrated environment according to the incremental integration testing.

Types of Incremental Integration Testing

Design hierarchy of a software can be seen in a tree-like structure, as shown in Fig. 7.8. In this tree-like structure, incremental integration can be done either from top to bottom or bottom to top. Based on this strategy, incremental integration testing is divided into two categories.

Top-down Integration Testing The strategy in top-down integration is to look at the design hierarchy from top to bottom. Start with the high-level modules and move downward through the design hierarchy.

Modules subordinate to the top module are integrated in the following two ways:

Depth first integration In this type, all modules on a major control path of the design hierarchy are integrated first. In the example shown in Fig. 7.8, modules 1, 2, 6, 7/8 will be integrated first. Next, modules 1, 3, 4/5 will be integrated.

Breadth first integration In this type, all modules directly subordinate at each level, moving across the design hierarchy horizontally, are integrated first. In the example shown in Fig. 7.8, modules 2 and 3 will be integrated first. Next, modules 6, 4, and 5 will be integrated. Modules 7 and 8 will be integrated last.

However, in practice, these two sequences of top-down integration cannot be used every time. In general, there is no best sequence, but the following guidelines can be considered:

1. In practice, the availability of modules matter the most. The module which is ready to be integrated, will be integrated and tested first. We should not wait to test it according to depth first or breadth first sequence, but use the availability of modules.
2. If there are critical sections of the software, design the sequence such that these sections will be added and tested as early as possible. A critical section might be a complex module, a module with a new algorithm or a module suspected to be error prone.
3. Design the sequence such that the I/O modules are added as early as possible so that all interface errors will be detected earlier.

Top-Down Integration Procedure

The procedure for top-down integration process is discussed in the following steps:

1. Start with the top or initial module in the software. Substitute the stubs for all the subordinate modules of top module. Test the top module.
2. After testing the top module, stubs are replaced one at a time with the actual modules for integration.
3. Perform testing on this recent integrated environment.
4. Regression testing may be conducted to ensure that new errors have not appeared.
5. Repeat steps 2–4 for the whole design hierarchy.

The top-down integration for Fig. 7.8 is shown below.

Listed below are the drawbacks of top-down integration testing.

1. Stubs must be prepared as required for testing one module.
2. Stubs are often more complicated than they first appear.
3. Before the I/O functions are added, the representation of test cases in stubs can be difficult.

Bottom-up Integration Testing

The bottom-up strategy begins with the terminal or modules at the lowest level in the software structure. After testing these modules, they are integrated and tested moving from bottom to top level. Since the processing required for modules subordinate to a given level is always available, stubs are not required in this strategy.

Bottom-up integration can be considered as the opposite of top-down approach. Unlike top-down strategy, this strategy does not require the architectural design of the system to be complete. Thus, bottom-up integration can be performed at an early stage in the developmental process. It may be used where the system reuses and modifies components from other systems.

The steps in bottom-up integration are as follows:

1. Start with the lowest level modules in the design hierarchy. These are the modules from which no other module is being called.
2. Look for the super-ordinate module which calls the module selected in step 1. Design the driver module for this super-ordinate module.
3. Test the module selected in step 1 with the driver designed in step 2.
4. The next module to be tested is any module whose subordinate modules (the modules it calls) have all been tested.
5. Repeat steps 2 to 5 and move up in the design hierarchy.
6. Whenever, the actual modules are available, replace stubs and drivers with the actual one and test again.

Bottom-up integration for Fig. 7.8, is shown below:

This type of integration is useful for integrating object-oriented systems, real-time systems, and systems with strict performance requirements. Bottom-up integration has the disadvantage that the software as a whole does not exist until the last module is added. It is also not an optimal strategy for functionally decomposed systems, as it tests the most important subsystem last.

Comparison between Top-Down and Bottom-Up Integration Testing

Table 7.1 provides a comparison between top-down and bottom-up integration testing techniques [29].

Table 7.1 Comparison between top-down and bottom-up testing

Issue	Top-Down Testing	Bottom-Up Testing
Architectural Design	It discovers errors in high-level design, thus detects errors at an early stage.	High-level design is validated at a later stage.
System Demonstration	Since we integrate the modules from top to bottom, the high-level design slowly expands as a working system. Therefore, feasibility of the system can be demonstrated to the top management.	It may not be possible to show the feasibility of the design. However, if some modules are already built as reusable components, then it may be possible to produce some kind of demonstration.
Test Implementation	*(nodes – 1)* stubs are required for the subordinate modules.	*(nodes – leaves)* test drivers are required for super-ordinate modules to test the lower-level modules.

Practical Approach for Integration Testing

There is no single strategy adopted for industry practice. For integrating the modules, one cannot rely on a single strategy. There are situations depending on the project in hand which will force to integrate the modules by combining top-down and bottom-up techniques. This combined approach is sometimes known as *sandwich integration testing*.

Selection of an integration testing strategy depends on software characteristics and sometimes project schedules. In general, sandwich testing strategy that uses top-down tests for upper levels of the program structure with bottom-up tests for subordinate levels is the best compromise.

The practical approach for adopting sandwich testing is driven by the following factors:

Priority There may be situations which force us to prioritize the modules to be integrated. Some of the prioritizing guidelines [7] are given below:

- Integration should follow the lines of first putting together those subsystems that are of great concern or with more important requirements.
- This prioritization might dictate top-down integration if the module has a high level of control on its subordinate modules.
- The modules with more user interfaces should be tested first, as they are more error-prone.
- If the customer is anxious to see a running program early in the testing phase, then top-down integration will help.
- In another situation, the machine interface and performance might be of special interest. Here, bottom-up integration would be dictated. With this approach, we stand better chance of experiencing a high degree of concurrency during our integration.
- The module whose cyclomatic complexity is high must be tested first.

Availability In practice, the availability of modules matter the most. The module which is ready to be integrated, will be integrated and tested first. We should not wait according to top-down or bottom-up sequence, but use the availability of modules. Different development schedules for different modules of the system force us to integrate with available modules only.

Pros and Cons of Decomposition-Based Integration

Decomposition-based integration techniques are better for monitoring the progress of integration testing along the decomposition tree. If there is failure in this integration, the first suspect goes to the recently added module, as the modules are integrated one by one either in top-down or bottom-up sequence. Thus, debugging is easy in decomposition-based integration.

However, there is more effort required in this type of integration, as stubs and drivers are needed for testing. Drivers are more complicated to design as compared to stubs. The integration testing effort is computed as the number of test sessions. A test session is one set of test cases for a specific configuration.

The total number of test sessions in a decomposition-based integration is computed as:

Number of test sessions = nodes − leaves + edges

7.2.2 Call Graph-Based Integration

It is assumed that integration testing detects bugs which are structural. However, it is also important to detect some behavioural bugs. If we can refine the functional decomposition tree into a form of module calling graph, then we are moving towards behavioural testing at the integration level. This can be done with the help of a *call graph*, as given by Jorgensen [78].

A call graph is a directed graph, wherein the nodes are either modules or units, and a directed edge from one node to another means one module has called another module. The call graph can be captured in a matrix form which is known as the *adjacency matrix*. For example, see Fig. 7.9, which is a call graph of a hypothetical program. The figure shows how one unit calls another. Its adjacency matrix is shown in Fig. 7.10. This matrix may help the testers a lot.

Figure 7.9 Example call graph

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	x	x	x	x					x	x									
2																			
3																			
4																			
5					x	x	x												
6																			
7																			
8								x							x				
9																x			
10												x	x						
11											x								
12																	x		
13																		x	
14														x					
15																			x
16																			
17																			
18																			
19																			
20																			

Figure 7.10 Adjacency matrix

The call graph shown in Fig. 7.9 can be used as a basis for integration testing. The idea behind using a call graph for integration testing is to avoid the efforts made in developing the stubs and drivers. If we know the calling sequence, and if we wait for the called or calling function, if not ready, then call graph-based integration can be used.

There are two types of integration testing based on call graph which are discussed next.

Pair-wise Integration

If we consider only one pair of calling and called modules, then we can make a set of pairs for all such modules, as shown in Fig. 7.11, for pairs 1–10 and 1–11. The resulting set will be the total test sessions which will be equal to the sum of all edges in the call graph. For example, in the call graph shown in Fig. 7.11, the number of test sessions is 19 which is equal to the number of edges in the call graph.

Figure 7.11 Pair-wise integration

Neighbourhood Integration

There is not much reduction in the total number of test sessions in pair-wise integration as compared to decomposition-based integration. If we consider the neighbourhoods of a node in the call graph, then the number of test sessions may reduce. The neighbourhood for a node is the immediate predecessor as well as the immediate successor nodes. The neighbourhood of a node, thus, can be defined as the set of nodes that are one edge away from the given node.

The neighbourhoods of each node in the call graph shown in Fig. 7.9 is shown in Table 7.2.

Table 7.2 Neighbourhood integration details

Node	Neighbourhoods	
	Predecessors	Successors
1	-----	2,3,4,5,10,11
5	1	6,7,8
8	5	9,16
9	8	17
10	1	13,14
11	1	12
12	11	18
13	10	19
14	10	15
15	14	20

The total test sessions in neighbourhood integration can be calculated as:

$$\begin{aligned}\textbf{Neighbourhoods} &= \textbf{nodes} - \textbf{sink nodes}\\ &= 20 - 10\\ &= 10\end{aligned}$$

where *sink node* is an instruction in a module at which the execution terminates.

7.2.3 Path-Based Integration

As we have discussed, in a call graph, when a module or unit executes, some path of source instructions is executed (remember flow graph?). And it may be possible that in that path execution, there may be a call to another unit. At that point, the control is transferred from the calling unit to the called unit. This passing of control from one unit to another unit is necessary for integration testing. Also, there should be information within the module regarding instructions that call the module or return to the module. This must be tested at the time of integration. It can be done with the help of path-based integration defined by Paul C. Jorgenson [78]. We need to understand the following definitions for path-based integration.

Source node It is an instruction in the module at which the execution starts or resumes. The nodes where the control is being transferred after calling the module are also source nodes.

Sink node It is an instruction in a module at which the execution terminates. The nodes from which the control is transferred are also sink nodes.

Module execution path (MEP) It is a path consisting of a set of executable statements within a module like in a flow graph.

Message When the control from one unit is transferred to another unit, then the programming language mechanism used to do this is known as a *message*. For example, when there is a function call, then it is a message from one unit (where the call is mentioned; caller module) to another unit (the unit which is being called).

MM-path It is a path consisting of MEPs and messages. The path shows the sequence of executable statements; it also crosses the boundary of a unit when a message is followed to call another unit. In other words, MM-path is a set of MEPs and transfer of control among different units in the form of messages.

MM-path graph It can be defined as an extended flow graph where nodes are MEPs and edges are messages. It returns from the last called unit to the first unit where the call was made. In this graph, messages are highlighted with thick lines.

Now let us see the concept of path-based integration with the help of one example. Fig. 7.12 shows the MM-path as a darken line. The details regarding the example units shown in Fig. 7.12 is given in Table 7.3.

Figure 7.12 MM-path

Table 7.3 MM-path details

	Source Nodes	Sink Nodes	MEPs
Unit A	1,4	3,5	MEP(A,1) = <1,2,5> MEP(A,2) = <1,3> MEP(A,3) = <4,5>
Unit B	1,5	4,6	MEP(B,1) = <1,2,4> MEP(B,2) = <5,6> MEP(B,3) = <1,2,3,4,5,6>
Unit C	1	5	MEP(C,1) = <1,3,4,5> MEP(C,2) = <1,2,4,5>

The MM-path graph for this example is shown in Fig. 7.13.

Figure 7.13 MEP graph

7.3 Function Testing

When an integrated system is tested, all its specified functions and external interfaces are tested on the software. Every functionality of the system specified in the functions is tested according to its external specifications. An external specification is a precise description of the software behaviour from the viewpoint of the outside world (e.g. user). Kit [1] has defined function testing as the *process of attempting to detect discrepancies between the functional specifications of a software and its actual behaviour.*

Thus, the objective of function test is to measure the quality of the functional (business) components of the system. Tests verify that the system behaves correctly from the user/business perspective and functions according to the requirements, models, or any other design paradigm used to specify the application. The function test must determine if each component or business event:

1. performs in accordance to the specifications,
2. responds correctly to all conditions that may present themselves by incoming events/data,
3. moves data correctly from one business event to the next (including data stores), and
4. is initiated in the order required to meet the business objectives of the system.

Function testing can be performed after unit and integration testing, or whenever the development team thinks that the system has sufficient functionality to execute some tests. The test cases are executed such that the execution of a given test case against the software will exercise external functionality of

certain parts. To keep a record of function testing, a function coverage metric is used. Function coverage can be measured with a *function coverage matrix*. It keeps track of those functions that exhibited the greatest number of errors. This information is valuable because it tells us that these functions probably contain the preponderance of errors that have not been detected yet.

An effective function test cycle must have a defined set of processes and deliverables. The primary processes/deliverables for requirements based function test are discussed below.

Test planning During planning, the test leader with assistance from the test team defines the scope, schedule, and deliverables for the function test cycle. He delivers a test plan (document) and a test schedule (work plan)—these often undergo several revisions during the testing cycle.

Partitioning/functional decomposition Functional decomposition of a system (or partitioning) is the breakdown of a system into its functional components or functional areas. Another group in the organization may take responsibility for the functional decomposition (or model) of the system, but the testing organization should still review this deliverable for completeness before accepting it into the test organization. If the functional decompositions or partitions have not been defined or are deemed insufficient, then the testing organization will have to take responsibility for creating and maintaining the partitions.

Requirement definition The testing organization needs specified requirements in the form of proper documents to proceed with the function test. These requirements need to be itemized under an appropriate functional partition.

Test case design A tester designs and implements a test case to validate that the product performs in accordance with the requirements. These test cases need to be itemized under an appropriate functional partition and mapped/traced to the requirements being tested.

Traceability matrix formation Test cases need to be traced/mapped back to the appropriate requirement. A function coverage matrix is prepared. This matrix is a table, listing specific functions to be tested, the priority for testing each function, and test cases that contain tests for each function. Once all the aspects of a function have been tested by one or more test cases, then the test design activity for that function can be considered complete. This approach gives a more accurate picture of the application when coverage analysis is done. For example, in Table 7.4, function F2 test cases must be executed, as its priority is highest; and through the function coverage matrix, we can track which functions are being tested through which test cases.

Table 7.4 Function coverage matrix

Functions/Features	Priority	Test Cases
F1	3	T2,T4,T6
F2	1	T1, T3,T5

Test case execution As in all phases of testing, an appropriate set of test cases need to be executed and the results of those test cases recorded. Which test cases are to be executed should be defined within the context of the test plan and the current state of the application being tested. If the current state of the application does not support the testing of one or more functions, then this testing should be deferred until it justifies the expenditure of testing resources.

7.4 System Testing

When all the modules have been integrated with the software system, then the software is incorporated with other system elements (e.g. hardware, people, information). This integrated system needs to be validated now. But the system is not validated according to functional specifications. System testing should not be misunderstood at this point. In function testing, we have already validated all the requirement functionality. Thus, system testing is not a process of testing the functions of a complete system or program.

System testing is the process of attempting to demonstrate that a program or system does not meet its original requirements and objectives, as stated in the requirement specification. System testing is actually a series of different tests to test the whole system on various grounds where bugs have the probability to occur. The ground can be performance, security, maximum load, etc. The integrated system is passed through various tests based on these grounds and depending on the environment and type of project. After passing through these tests, the resulting system is a system which is ready for acceptance testing which involves the user, as shown in Fig. 7.14.

But it is difficult to test the system on various grounds, since there is no methodology for system testing. One solution to this problem is to think from the perspective of the user and the problem the user is trying to solve.

Unlike function testing, the external specifications cannot be used as the basis for deriving the system test cases, since this would weaken the purpose of system testing. On the other hand, the objectives document cannot be used by itself to formulate test cases, since it does not by definition, contain precise

Figure 7.14 System testing

descriptions of the program's external interfaces. This dilemma is solved by using the program's user documentation. Design the system test by analysing the objectives; formulate test cases by analysing the user documentation. This has a useful side-effect of comparing the program with its objectives and the user documentation, as well as comparing the user documentation with the objectives, as shown in Fig. 7.15.

It is obvious that the central purpose of system testing is to compare the system with its stated objectives. However, there are no test case design methodologies. The reason for this is that objectives state what a program should do and how well the program should do it, but they do not state the representation of the program's functions.

Figure 7.15 Design the system by analysing the objectives

Given the statement of objectives, there is no identifiable methodology that would yield a set of test cases, other than the vague but useful guideline of writing test cases to attempt to show that the system is inconsistent with each sentence in the objectives statement. Therefore, rather than developing a system test methodology, distinct categories of system test cases are taken. Some of the categories are discussed below.

7.4.1 Categories of System Tests

Recovery Testing

Recovery is just like the exception handling feature of a programming language. It is the ability of a system to restart operations after the integrity of the application has been lost. It reverts to a point where the system was functioning correctly and then, reprocesses the transactions to the point of failure.

Some software systems (e.g. operating system, database management systems, etc.) must recover from programming errors, hardware failures, data errors, or any disaster in the system. So the purpose of this type of system testing is to show that these recovery functions do not work correctly.

The main purpose of this test is to determine how good the developed software is when it faces a disaster. Disaster can be anything from unplugging the system which is running the software from power, network etc., also stopping the database, or crashing the developed software itself. Thus, *recovery testing is the activity of testing how well the software is able to recover from crashes, hardware failures, and other similar problems*. It is the forced failure of the software in various ways to verify that the recovery is properly performed. Some examples of recovery testing are given below:

- While the application is running, suddenly restart the computer and thereafter, check the validity of application's data integrity.
- While the application receives data from the network, unplug the cable and plug-in after awhile, and analyse the application's ability to continue receiving data from that point, when the network connection disappeared.
- Restart the system while the browser has a definite number of sessions and after rebooting, check that it is able to recover all of them.

Recovery tests would determine if the system can return to a well-known state, and that no transactions have been compromised. Systems with automated recovery are designed for this purpose. There can be provision of multiple CPUs and/or multiple instances of devices, and mechanisms to detect the failure of a device. A 'checkpoint' system can also be put that meticulously records transactions and system states periodically to preserve them in case of failure. This information allows the system to return to a known state after the failure.

Beizer [49] proposes that testers should work on the following areas during recovery testing:

Restart If there is a failure and we want to recover and start again, then first the current system state and transaction states are discarded. Following the criteria of checkpoints as discussed above, the most recent checkpoint record

is retrieved and the system is initialized to the states in the checkpoint record. Thus, by using checkpoints, a system can be recovered and started again from a new state. Testers must ensure that all transactions have been reconstructed correctly and that all devices are in proper states. The system now is in a position to begin to process new transactions.

Switchover Recovery can also be done if there are standby components and in case of failure of one component, the standby takes over the control. The ability of the system to switch to a new component must be tested.

A good way to perform recovery testing is under maximum load. Maximum load would give rise to transaction inaccuracies and the system would crash, resulting in defects and design flaws.

Security Testing

Safety and security issues are gaining importance due to the proliferation of commercial applications on the Internet and the increasing concern about privacy. Security is a protection system that is needed to assure the customers that their data will be protected. For example, if Internet users feel that their personal data/information is not secure, the system loses its accountability. Security may include controlling access to data, encrypting data in communication, ensuring secrecy of stored data, auditing security events, etc. The effects of security breaches could be extensive and can cause loss of information, corruption of information, misinformation, privacy violations, denial of service, etc.

Types of Security Requirements While performing security testing, the following security requirements must be considered:

- Security requirements should be associated with each functional requirement. Each functional requirement, most likely, has a specific set of related security issues to be addressed in the software implementation. For example, the log-on requirement in a client-server system must specify the number of retries allowed, the action to be taken if the log-on fails, and so on.
- In addition to security concerns that are directly related to particular requirements, a software project has security issues that are global in nature, and are therefore, related to the application's architecture and overall implementation. For example, a Web application may have a global requirement that all private customer data of any kind is stored in encrypted form in the database. In another example, a system-wide security requirement is to use SSL to encrypt the data sent between the client browser and the Web server. Security testing team must verify that SSL is correctly used in all such transmissions.

Security testing is the process of attempting to devise test cases to evaluate the adequacy of protective procedures and countermeasures.

The problem with security testing is that security-related bugs are not as obvious as other type of bugs. It may be possible that the security system has failed and caused the loss of information without the knowledge of loss. Thus, the tester should perform security testing with the goal to identify the bugs that are very difficult to identify.

Security vulnerabilities Vulnerability is an error that an attacker can exploit. Security vulnerabilities are of the following types:

- Bugs at the implementation level, such as local implementation errors or interprocedural interface errors
- Design-level mistakes

Design-level vulnerabilities are the hardest defect category to handle, but they are also the most prevalent and critical. Unfortunately, ascertaining whether a program has design-level vulnerabilities requires great expertise, which makes finding not only difficult but particularly hard to automate. Examples of design-level security flaws include problem in error-handling, unprotected data channels, incorrect or missing access control mechanisms, and timing errors especially in multithreaded systems.

How to perform security testing Testers must use a risk-based approach, grounded in both the system's architectural reality and the attacker's mindset, to gauge software security adequately. By identifying risks and potential loss associated with those risks in the system and creating tests driven by those risks, the tester can properly focus on areas of code in which an attack is likely to succeed. Therefore, risk analysis, especially at the design-level, can help us identify potential security problems and their impacts. Once identified and ranked, software risks can help guide software security testing.

Risk management and security testing Software security practitioners perform many different tasks to manage software security risks, including:

- Creating security abuse/misuse cases
- Listing normative security requirements
- Performing architectural risk analysis
- Building risk-based security test plans
- Wielding static analysis tools
- Performing security tests

Three tasks, i.e. architectural risk analysis, risk-based security test planning, and security testing, are closely linked because a critical aspect of security

testing relies on probing security risks. Based on design-level risk analysis and ranking of security related risks, security test plans are prepared which guide the security testing.

Thus, security testing must necessarily involve two diverse approaches:

- Testing security mechanisms to ensure that their functionality is properly implemented
- Performing risk-based security testing motivated by understanding and simulating the attacker's approach

Elements of security testing The basic security concepts that need to be covered by security testing are discussed below:

Confidentiality A security measure which protects against the disclosure of information to parties other than the intended recipient.

Integrity A measure intended to allow the receiver to determine that the information which it receives has not been altered in transit or by anyone other than the originator of the information. Integrity schemes often use some of the same underlying technologies as confidentiality schemes, but they usually involve adding additional information to a communication to form the basis of an algorithmic check rather than encoding all the communication.

Authentication A measure designed to establish the validity of a transmission, message, or originator. It allows the receiver to have confidence that the information it receives originates from a specific known source.

Authorization It is the process of determining that a requester is allowed to receive a service or perform an operation. Access control is an example of authorization.

Availability It assures that the information and communication services will be ready for use when expected. Information must be kept available for authorized persons when they need it.

Non-repudiation A measure intended to prevent the later denial that an action happened, or a communication took place, etc. In communication terms, this often involves the interchange of authentication information combined with some form of provable timestamp.

Performance Testing

Performance specifications (requirements) are documented in a performance test plan. Ideally, this is done during the requirements development phase of any system development project, prior to any design effort.

Performance specifications (requirements) should ask the following questions, at a minimum:

- In detail, what is the performance test scope? What subsystems, interfaces, components, etc. are in and out of the scope for this test?
- For the user interfaces (UIs) involved, how many concurrent users are expected for each (specify peak vs. nominal)?
- What does the target system (hardware) look like (specify all server and network appliance configurations)?
- What are the time requirements for any/all backend batch processes (specify peak vs. nominal)?

A system along with its functional requirements, must meet the quality requirements. One example of quality requirement is performance level. The users may have objectives for a software system in terms of memory use, response time, throughput, and delays. Thus, performance testing is to test the run-time performance of the software on the basis of various performance factors. Performance testing becomes important for real-time embedded systems, as they demand critical performance requirements.

This testing requires that performance requirements must be clearly mentioned in the SRS and system test plans. The important thing is that these requirements must be quantified. For example, a requirement that the system should return a response to a query in a reasonable amount of time is not an acceptable requirement; the time must be specified in a quantitative way. In another example, for a Web application, you need to know at least two things: (a) expected load in terms of concurrent users or HTTP connections and (b) acceptable response time.

Performance testing is often used as a part of the process of performance profile tuning. The goal is to identify the 'weakest links'—the system often carries a number of parts which, with a little tweak, can significantly improve the overall performance of the system. It is sometimes difficult to identify which parts of the system represent the critical paths. To help identify critical paths, some test tools include (or have as add-ons) instrumentation agents that run on the server and report transaction times, database access times, network overhead, and other server monitors. Without such instrumentation, the use of primitive system tools may be required (e.g. Task Manager in Microsoft Windows). Performance testing usually concludes that it is the software (rather than the hardware) that contribute most to delays (bottlenecks) in data processing.

For performance testing, generally a generic set of sample data is created in the project and because of time constraints, that data is used throughout

development and functional testing. However, these datasets tend to be unrealistic, with insufficient size and variation, which may result in performance surprises when an actual customer, using much more data than contained in the sample dataset, attempts to use the system. Therefore, it becomes necessary that testing team must use production-size realistic databases that include a wide range of possible data combinations.

Using realistic-size databases provides the following benefits:

- Since a large dataset may require significant disk space and processor power, it leads to backup and recoverability concerns.
- If data is transferred across a network, bandwidth may also be a consideration.

Thus, working with realistic datasets as early as possible will help bring these issues to the surface while there is time and when it is more cost-effective to do something about them, rather than after the software is already in production environment, when such issues can become very costly to address in terms of both budget and schedule.

The following tasks must be done for this testing:

- Decide whether to use internal or external resources to perform tests, depending on in-house expertise (or the lack of it).
- Gather performance requirements (specifications) from users and/or business analysts.
- Develop a high-level plan (or project charter), including requirements, resources, timelines, and milestones.
- Develop a detailed performance test plan (including detailed scenarios and test cases, workloads, environment info, etc).
- Choose test tool(s).
- Specify test data needed.
- Develop detailed performance test project plan, including all dependencies and associated timelines.
- Configure the test environment (ideally identical hardware to the production platform), router configuration, deployment of server instrumentation, database test sets developed, etc.
- Execute tests, probably repeatedly (iteratively), in order to see whether any unaccounted factor might affect the results.

Load Testing

Normally, we don't test the system with its full load, i.e. with maximum values of all resources in the system. The normal system testing takes into consider-

ation only nominal values of the resources. However, there is high probability that the system will fail when put under maximum load. Therefore, it is important to test the system with all its limits. *When a system is tested with a load that causes it to allocate its resources in maximum amounts, it is called load testing.* The idea is to create an environment more demanding than the application would experience under normal workloads.

When you are doing performance testing, it may not be possible that you are taking all statistically representative loads for all the parameters in the system. For example, the system is designed for n number of users, t number of transactions per user per unit time, with p unit time response time, etc. Have you tested the performance of the system with all these representative loads? If no, then load testing must be performed as a part of performance testing. Load is varied from a minimum (zero) to the maximum level the system can sustain without running out of resources. As the load is being increased, transactions may suffer (application-specific) excessive delay. For example, when many users work simultaneously on a web server, the server responds slow. In this way, through load testing, we are able to determine the maximum sustainable load the system can handle.

Stress Testing

Stress testing is also a type of load testing, but the difference is that the system is put under loads beyond the limits so that the system breaks. Thus, *stress testing tries to break the system under test by overwhelming its resources in order to find the circumstances under which it will crash.* The areas that may be stressed in a system are:

- Input transactions
- Disk space
- Output
- Communications
- Interaction with users

Stress testing is important for real-time systems where unpredictable events may occur, resulting in input loads that exceed the values described in the specification, and the system cannot afford to fail due to maximum load on resources. Therefore, in real-time systems, the entire threshold values and system limits must be noted carefully. Then, the system must be stress-tested on each individual value.

Stress testing demands the amount of time it takes to prepare for the test and the amount of resources consumed during the actual execution of the test. Therefore, stress testing cost must be weighed against the risk of not identify-

ing volume-related failures. For example, many of the defense systems which are real-time in nature, demand the systems to perform continuously in warfare conditions. Thus, for a real-time defense system, we must stress-test the system; otherwise, there may be loss of equipment as well as life.

Usability Testing

This type of system testing is related to a system's presentation rather than its functionality. System testing can be performed to find human-factors or usability problems on the system. The idea is to adapt the software to users' actual work styles rather than forcing the users to adapt according to the software. Thus, the goal of usability testing is to verify that intended users of the system are able to interact properly with the system while having a positive and convenient experience. *Usability testing identifies discrepancies between the user interfaces of a product and the human engineering requirements of its potential users.*

What the user wants or expects from the system can be determined using several ways as proposed by Dustin [12]:

Area experts The usability problems or expectations can be best understood by the subject or area experts who have worked for years in the same area. They analyse the system's specific requirements from the user's perspective and provide valuable suggestions.

Group meetings Group meeting is an interesting idea to elicit usability requirements from the user. These meetings result in potential customers' comments on what they would like to see in an interface.

Surveys Surveys are another medium to interact with the user. It can also yield valuable information about how potential customers would use a software product to accomplish their tasks.

Analyse similar products We can also analyse the experiences in similar kinds of projects done previously and use it in the current project. This study will also give us clues regarding usability requirements.

Usability characteristics against which testing is conducted are discussed below:

Ease of use The users enter, navigate, and exit with relative ease. Each user interface must be tailored to the intelligence, educational background, and environmental pressures of the end-user.

Interface steps User interface steps should not be misleading. The steps should also not be very complex to understand either.

Response time The time taken in responding to the user should not be so high that the user is frustrated or will move to some other option in the interface.

Help system A good user interface provides help to the user at every step. The help documentation should not be redundant; it should be very precise and easily understood by every user of the system.

Error messages For every exception in the system, there must be an error message in text form so that users can understand what has happened in the system. Error messages should be clear and meaningful.

An effective tool in the development of a usable application is the user-interface prototype. This kind of prototype allows interaction between potential users, requirements personnel, and developers to determine the best approach to the system's interface. Prototypes are far superior because they are interactive and provide a more realistic preview of what the system will look like.

Compatibility/Conversion/Configuration Testing

Compatibility testing is to check the compatibility of a system being developed with different operating system, hardware and software configuration available, etc. Many software systems interact with multiple CPUs. Some software control real-time process and embedded software interact with devices. In many cases, users require that the devices be interchangeable, removable, or re-configurable. Very often the software will have a set of commands or menus that allow users to make these configuration changes. Configuration testing allows developers/testers to evaluate system performance and availability when hardware exchanges and reconfigurations occur. The number of possible combinations for checking the compatibilities of available hardware and software can be too high, making this type of testing a complex job. Discussed below are some guidelines for compatibility testing [12].

Operating systems The specifications must state all the targeted end-user operating systems on which the system being developed will be run.

Software/Hardware The product may need to operate with certain versions of Web browsers, with hardware devices such as printers, or with other softwares such as virus scanners or word processors.

Conversion testing Compatibility may also extend to upgrades from previous versions of the software. Therefore, in this case, the system must be upgraded properly and all the data and information from the previous version should also be considered. It should be specified whether the new system will be backward compatible with the previous version. Also, if other user's preferences or settings are to be preserved or not.

Ranking of possible configurations Since there will be a large set of possible configurations and compatibility concerns, the testers must rank the possible configurations in order, from the most to the least common, for the target system.

Identification of test cases Testers must identify appropriate test cases and data for compatibility testing. It is usually not feasible to run the entire set of possible test cases on every possible configuration, because this would take too much time. Therefore, it is necessary to select the most representative set of test cases that confirms the application's proper functioning on a particular platform.

Updating the compatibility test cases The compatibility test cases must also be continually updated with the following:

(i) Tech-support calls provide a good source of information for updating the compatibility test suite.

(ii) A beta-test program can also provide a large set of data regarding real end-user configurations and compatibility issues prior to the final release of a system.

7.5 Acceptance Testing

Developers/testers must keep in mind that the software is being built to satisfy the user requirements and no matter how elegant its design is, it will not be accepted by the users unless it helps them achieve their goals as specified in the requirements. After the software has passed all the system tests and defect repairs have been made, the customer/client must be involved in the testing with proper planning. The purpose of acceptance testing is to give the end-user a chance to provide the development team with feedback as to whether or not the software meets their needs. Ultimately, it's the user that needs to be satisfied with the application, not the testers, managers, or contract writers.

Acceptance testing is one of the most important types of testing we can conduct on a product. It is more important to worry whether users are happy about the way a program works rather than whether or not the program passes a bunch of tests that were created by testers in an attempt to validate the requirements, that an analyst did their best to capture and a programmer interpreted based on their understanding of those requirements.

Thus, *acceptance testing is the formal testing conducted to determine whether a software system satisfies its acceptance criteria and to enable buyers to determine whether to accept the system or not.*

Acceptance testing must take place at the end of the development process. It consists of tests to determine whether the developed system meets the prede-

termined functionality, performance, quality, and interface criteria acceptable to the user. Therefore, the final acceptance acknowledges that the entire software product adequately meets the customer's requirements. User acceptance testing is different from system testing. System testing is invariably performed by the development team which includes developers and testers. User acceptance testing, on the other hand, should be carried out by end-users.

Thus, acceptance testing is designed to:

- Determine whether the software is fit for the user.
- Making users confident about the product.
- Determine whether a software system satisfies its acceptance criteria.
- Enable the buyer to determine whether to accept the system or not.

The final acceptance marks the completion of the entire development process. It happens when the customer and the developer has no further problems.

Acceptance test might be supported by the testers. It is very important to define the acceptance criteria with the buyer during various phases of SDLC. A well-defined acceptance plan will help development teams to understand users' needs. The acceptance test plan must be created or reviewed by the customer. The development team and the customer should work together and make sure that they:

- Identify interim and final products for acceptance, acceptance criteria, and schedule.
- Plan how and by whom each acceptance activities will be performed.
- Schedule adequate time for the customer to examine and review the product.
- Prepare the acceptance plan.
- Perform formal acceptance testing at delivery.
- Make a decision based on the results of acceptance testing.

Entry Criteria

- System testing is complete and defects identified are either fixed or documented.
- Acceptance plan is prepared and resources have been identified.
- Test environment for the acceptance testing is available.

Exit Criteria

- Acceptance decision is made for the software.
- In case of any warning, the development team is notified.

Types of Acceptance Testing

Acceptance testing is classified into the following two categories:

- *Alpha Testing* Tests are conducted at the development site by the end users. The test environment can be controlled a little in this case.
- *Beta Testing* Tests are conducted at the customer site and the development team does not have any control over the test environment.

7.5.1 Alpha Testing

Alpha is the test period during which the product is complete and usable in a test environment, but not necessarily bug-free. It is the final chance to get verification from the customers that the tradeoffs made in the final development stage are coherent.

Therefore, alpha testing is typically done for two reasons:

(i) to give confidence that the software is in a suitable state to be seen by the customers (but not necessarily released).

(ii) to find bugs that may only be found under operational conditions. Any other major defects or performance issues should be discovered in this stage.

Since alpha testing is performed at the development site, testers and users together perform this testing. Therefore, the testing is in a controlled manner such that if any problem comes up, it can be managed by the testing team.

Entry Criteria to Alpha

- All features are complete/testable (no urgent bugs).
- High bugs on primary platforms are fixed/verified.
- 50% of medium bugs on primary platforms are fixed/verified.
- All features have been tested on primary platforms.
- Performance has been measured/compared with previous releases (user functions).
- Usability testing and feedback (ongoing).
- Alpha sites are ready for installation.

Exit Criteria from Alpha

After alpha testing, we must:

- Get responses/feedbacks from customers.
- Prepare a report of any serious bugs being noticed.
- Notify bug-fixing issues to developers.

7.5.2 Beta Testing

Once the alpha phase is complete, development enters the beta phase. Beta is the test period during which the product should be complete and usable in a production environment. The purpose of the beta ship and test period is to test the company's ability to deliver and support the product (and not to test the product itself). Beta also serves as a chance to get a final 'vote of confidence' from a few customers to help validate our own belief that the product is now ready for volume shipment to all customers.

Versions of the software, known as beta-versions, are released to a limited audience outside the company. The software is released to groups of people so that further testing can ensure the product has few or no bugs. Sometimes, beta-versions are made available to the open public to increase the feedback field to a maximal number of future users.

Testing during the beta phase, informally called beta testing, is generally constrained to black-box techniques, although a core of test engineers are likely to continue with white-box testing parallel to beta tests. Thus, the term *beta test* can refer to a stage of the software–closer to release than being 'in alpha'–or it can refer to the particular group and process being done at that stage. So a tester might be continuing to work in white-box testing while the software is 'in beta' (a stage), but he or she would then not be a part of 'the beta test' (group/activity).

Entry Criteria to Beta

- Positive responses from alpha sites.
- Customer bugs in alpha testing have been addressed.
- There are no fatal errors which can affect the functionality of the software.
- Secondary platform compatibility testing is complete.
- Regression testing corresponding to bug fixes has been done.
- Beta sites are ready for installation.

Guidelines for Beta Testing

- Don't expect to release new builds to beta testers more than once every two weeks.
- Don't plan a beta with fewer than four releases.
- If you add a feature, even a small one, during the beta process, the clock goes back to the beginning of eight weeks and you need another 3–4 releases.

Exit Criteria from Beta

After beta testing, we must:

- Get responses/feedbacks from the beta testers.
- Prepare a report of all serious bugs.
- Notify bug-fixing issues to developers.

SUMMARY

This chapter describes all the validation testing activities. The test plans for all validation testing types are done in advance while doing verification at every stage of SDLC. Validation testing is done at three levels: unit, integration of all modules, and system. Unit validation cannot be done in isolation. It is performed with the help of drivers and stubs which increase the cost of the project. But they are unavoidable if testing has to be performed of a module that gets input from another module which is not ready, or that calls other modules which are not ready.

The next level is integration of all the unit tested modules and then test the integrated modules. Validation testing continues at the system level which has various types depending upon the project, user requirements, etc. The integrated system must be tested on these grounds to test its performance, functionality, etc.

The final tested system then goes to another level of validation, i.e. testing the system with the user so that he accepts it according to his satisfaction. This is known as acceptance testing. Acceptance testing is performed at the developer's site as well as the customer's own site. When the customer is satisfied at his own site, then the final system is installed in the operation status. All these validation activities have been discussed in detail in this chapter.

These validation testing activities are necessary to produce a better system. If we do not perform level-wise validation testing, the system will have many bugs which are also difficult to find. But if we test the system at every level, we reduce the chances of bugs. For example, if we perform unit validation, we test every module's functionality. When we integrate the modules and test them, we uncover the interface bugs, if they are present. Similarly, when we perform system testing on various grounds like security, stress, performance, we try to make a system which will work in severe conditions as a bug-proof system.

Let us review the important concepts described in this chapter:

- Unit testing ensures that the software meets at least a baseline level of functionality at a module-level prior to integration and system testing.
- A test driver can be defined as a software module which is used to invoke a module under test and provide test inputs, control and monitor execution, and report test results or most simplistically, a line of code that calls a method and passes that method a value.
- A stub can be defined as a piece of software that works similar to a unit which is referenced by the unit being tested, but it is much simpler than the actual unit. It works as a 'stand-in' for the subordinate unit and provides the minimum required behaviour for that unit.
- Integration testing focuses on bugs caused by interfacing between the modules while integrating them.

- Various types of integration testing can be seen in a hierarchical tree given below:

- Decomposition-based integration is based on the decomposition of design into functional components or modules in a hierarchical tree form consisting of modules as nodes and interfaces between them as edges between those nodes. Thus, the interfaces among separately tested modules are tested.
- In non-incremental integration testing, either all untested modules are combined together and then tested, or unit-tested modules are combined together.
- In incremental integration testing, start with one module and unit-test it. Then combine the module which has to be merged with it and test both the modules. In this way, incrementally keep on adding the modules and test the recent environment.
- The strategy in top-down integration is to look at the design hierarchy from top to bottom. Start with high-level modules and move downward through the design hierarchy.
- The bottom-up strategy begins with terminal nodes or modules at the lowest level in the software structure. After testing these modules, they are integrated and tested, moving from bottom to top level.
- The total test sessions in a decomposition-based integration is computed as:

  Number of test sessions = nodes – leaves + edges

- Call graph-based integration uses call graphs, wherein the nodes are either modules or units, and a directed edge from one node to another means one module has called another module. The graph and its corresponding adjacency matrix help the testers in integration testing.
- Function testing is the process of attempting to detect discrepancies between the functional specifications of a software and its actual behaviour.
- System testing is the process of attempting to demonstrate that a program or system

does not meet its original requirements and objectives, as stated in the requirement specification.

- Recovery is the ability of a system to restart operations after the integrity of the application has been lost.
- Security is a protection system that is needed to secure confidential information and to assure the customer that their data will be protected.
- Stress testing tries to break the system under test by overwhelming its resources and finding the circumstances under which it will crash.
- Performance testing is to test the system against all performance requirements mentioned in the SRS. It is often used as a part of performance profile tuning. The goal is to identify the 'weakest links' — the system often carries a number of parts which, with a little tweak, can significantly improve the overall performance of the system.
- Usability testing identifies discrepancies between the user interfaces of a product and the human engineering requirements of its potential users.
- Compatibility testing is to check the compatibility of a system being developed with different operating systems, hardware and software configurations available, etc.
- Configuration testing allows developers/testers to evaluate system performance and availability, when hardware exchanges and reconfigurations occur.
- Acceptance testing is the formal testing conducted to determine whether a software system satisfies its acceptance criteria and to enable buyers to determine whether to accept the system or not. It is of two types: alpha testing and beta testing.
- Alpha testing is performed with customers at the developers' site.
- Beta testing is performed by customers at their own site.

Exercises

Multiple Choice Questions

1. Software validation is achieved through a series of _____ tests that demonstrate conformity with requirements.
   (a) white-box
   (b) black-box
   (c) unit tests
   (d) none of the above
2. Before we validate the entire software, _____ must be validated first.
   (a) modules
   (b) system
   (c) functionality
   (d) all of the above
3. Unit tests ensure that the software meets at least a _____ of functionality prior to integration and system testing.
   (a) high-level
   (b) low-level

(c) baseline level
(d) none of the above

4. Two types of interface modules which must be simulated, if required, to test the module are _____.
   (a) unit and integration
   (b) simulators and emulators
   (c) stubs and drivers
   (d) none of the above
5. Overhead of stubs and drivers may increase the _____ of the entire software system.
   (a) test cases
   (b) time
   (c) cost
   (d) time and cost
6. Integration of modules is according to the _____ of software.
   (a) design
   (b) coding
   (c) specifications
   (d) all of the above
7. Recovery is the ability of a system to _____ operations after the integrity of the application has been lost.
   (a) suspend
   (b) observe
   (c) restart
   (d) all of the above
8. A system that meticulously records transactions and system states periodically so that these are preserved in case of a failure is called a _____.
   (a) checkpoint
   (b) transaction system
   (c) recovery system
   (d) none of the above
9. Security requirements should be associated with each _____ requirement.
   (a) functional
   (b) design
   (c) coding
   (d) testing
10. Measures intended to allow the receiver to determine that the information which it receives has not been altered in transit is known as _____.
   (a) confidentiality
   (b) integrity

(c) authentication
(d) none of the above

11. The process of determining that a requester is allowed to receive a service or perform an operation is called _____.
    (a) confidentiality
    (b) integrity
    (c) authentication
    (d) authorization
12. A measure intended to prevent the later denial that an action happened, or a communication took place is called _____.
    (a) confidentiality
    (b) integrity
    (c) non-repudiation
    (d) authorization
13. The type of system testing related to a system's presentation rather than its functionality is called _____.
    (a) usability testing
    (b) stress testing
    (c) conversion testing
    (d) all of the above
14. A system test should not be performed by _____.
    (a) programmers
    (b) testers
    (c) designers
    (d) all of the above
15. Acceptance testing must occur at the _____ of the development process.
    (a) start
    (b) end
    (c) middle
    (d) none of the above
16. Top-down integration testing requires _____ stubs.
    (a) nodes + 2
    (b) nodes – 1
    (c) nodes + 1
    (d) none
17. Bottom-up integration testing requires _____ drivers.
    (a) nodes + 2
    (b) nodes + leaves
    (c) nodes – leaves
    (d) none

18. Total number of sessions in decomposition-based integration testing is _____.
    (a) nodes + 2
    (b) nodes + leaves
    (c) nodes – leaves + edges
    (d) none of the above
19. The total number of sessions in a pair-wise call graph-based integration testing is _____.
    (a) total edges in the graph + 2
    (b) nodes + leaves
    (c) nodes – leaves + edges
    (d) total edges in the graph
20. Total number of sessions in neighbourhood call graph-based integration testing is _____.
    (a) total edges in the graph + 2
    (b) nodes + sink nodes
    (c) nodes – leaves + edges
    (d) nodes – sink nodes
21. The nodes where the control is being transferred after calling the module, are called _____.
    (a) sink nodes
    (b) source nodes
    (c) message
    (d) none of the above
22. The nodes from which the control is transferred are called _____.
    (a) sink nodes
    (b) source nodes
    (c) message
    (d) none of the above
23. When the control from one unit is transferred to another unit, then the programming language mechanism used to do this is known as
    (a) sink nodes
    (b) source nodes
    (c) message
    (d) none of the above
24. A call graph is a _____.
    (a) undirected graph
    (b) cyclic graph
    (c) directed graph
    (d) none of the above

Review Questions

1. Consider Problem 5.12. (a) Suppose main() module is not ready for the testing of CheckPrime() module. Design driver module for main(). (b) Module CheckPrime() is not ready when called in main(). Design stubs for this module.
2. Perform top-down and bottom-up integration procedure from the following system hierarchy.

3. Calculate the number of test sessions for the decomposition tree shown in Problem 2.
4. Consider Problem 5.12 again. Perform path-based integration calculating source nodes, sink nodes, MEPs and MM-path graph.
5. What are the factors that guide sandwich integration testing?
6. 'Design the system test by analysing the objectives.' Comment on this statement.
7. What is recovery testing? Give some examples.
8. What is risk-based security testing?
9. What are the elements of security testing?
10. What is the importance of realistic size databases in performance testing?
11. What is the difference between alpha and beta testing?
12. What is the entry and exit criteria for alpha and beta testing?
13. Calculate the number of test sessions for the pair-wise and neighbourhood call graph-based integration testing for the call graph shown below.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

8

Regression Testing

OBJECTIVES

After reading this chapter, you should be able to understand:

- Regression testing is necessary to maintain software whenever it is updated
- Regression testing increases the quality of software
- Objectives of regression testing
- Types of regression testing
- Regression testability
- Problems of regression testing
- Regression testing techniques

In Chapter 7, integration testing was discussed as one of the validation activity. However, when a new module is added as a part of integration testing, the software changes. New data flow paths are established, new I/O may occur, etc. These changes, due to addition of new modules, may affect the functioning of earlier integrated modules which otherwise were functioning flawlessly. Moreover, the software may also change whenever a new bug in the working system appears and it needs some changes. The new modifications may affect other parts of the software too. It means that whenever the software configuration changes, there is a probability of other modules getting affected due to bug fixing. In this case, it becomes important to test the whole software again with all the test cases so that a new modification in a certain part of the software does not affect other parts of the software.

8.1 Progressive vs. Regressive Testing

Whatever test case design methods or testing techniques, discussed until now, have been referred to as progressive testing or development testing. From verification to validation, the testing process progresses towards the release of the product. However, to maintain the software, bug fixing may be required during any stage of development and therefore, there is a need to check the software again to validate that there has been no adverse effect on the already working software. A system under test (SUT) is said to *regress* if

- a modified component fails, or
- a new component, when used with unchanged components, causes failures in the unchanged components by generating side-effects or feature interactions.

Therefore, now the following versions will be there in the system:

Baseline version The version of a component (system) that has passed a test suite.

Delta version A changed version that has not passed a regression test.

Delta build An executable configuration of the SUT that contains all the delta and baseline components.

Thus, it can be said that most test cases begin as progressive test cases and eventually become regression test cases. Regression testing is not another testing activity. Rather, it is the re-execution of some or all of the already developed test cases.

Definition

The purpose of regression testing is to ensure that bug-fixes and new functionalities introduced in a new version of the software do not adversely affect the correct functionality inherited from the previous version. IEEE software glossary defines regression testing as follows [44]:

Regression testing is the selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.

After a program has been modified, we must ensure that the modifications work correctly and check that the unmodified parts of the program have not been adversely affected by these modifications. It may be possible that small changes in one part of a program may have subtle undesired effects on other unrelated modules of the software. It is not necessary that if you are getting the desired outputs on the screen, then there is no problem. It may produce incorrect outputs on other test cases on which the original software produced correct outputs earlier. Thus, during regression testing, the modified software is executed on all tests to validate that it still behaves the same as it did in the original software, except where a change is expected.

Thus, regression testing can be defined as the *software maintenance task performed on a modified program to instill confidence that changes are correct and have not adversely affected the unchanged portions of the program.*

8.2 Regression Testing produces Quality Software

As discussed before, regression testing is performed in case of bug-fixing or whenever there is a need to incorporate any new requirements in the software. After the first release, whenever there is a need to fix the bugs that have

been reported or if there is any updation in the requirement itself, the developer fixes the bug or incorporates the new requirement by changing the code somewhere. This changed software requires to be fully tested again so as to ensure: (a) bug-fixing has been carried out successfully, (b) the modifications have not affected other unchanged modules, and (c) the new requirements have been incorporated correctly. (see Fig. 8.1). It means that every time you make a change in the software for whatever reasons, you have to perform a set of regression tests to validate the software. This process of executing the whole set of test again and again may be time-consuming and frustrating, but it increases the quality of software. Therefore, the creation, maintenance, and execution of a regression test suite helps to retain the quality of the software. Industry experiences noted by Onoma *et al.* [45] indicate that regression testing often has a strong positive influence on software quality. Indeed, the importance of regression testing is well-understood for the following reasons:

- It validates the parts of the software where changes occur.
- It validates the parts of the software which may be affected by some changes, but otherwise unrelated.
- It ensures proper functioning of the software, as it was before changes occurred.
- It enhances the quality of software, as it reduces the risk of high-risk bugs.

Figure 8.1 Regression testing produces Quality Software

8.3 Regression Testability

Regression testability refers to the property of a program, modification, or test suite that lets it be effectively and efficiently regression-tested. Leung and White [47] classify a program as regression testable if most single statement

modifications to the program entail rerunning a small proportion of the current test suite. Under this definition, regression testability is a function of both the design of the program and the test suite.

To consider regression testability, a regression number is computed. It is the average number of affected test cases in the test suite that are affected by any modification to a single instruction. This number is computed using information about the test suite coverage of the program.

If regression testability is considered at an early stage of development, it can provide significant savings in the cost of development and maintenance of the software.

8.4 Objectives of Regression Testing

It tests to check that the bug has been addressed The first objective in bug-fix testing is to check whether the bug-fixing has worked or not. Therefore, you run exactly the same test that was executed when the problem was first found. If the program fails on this testing, it means the bug has not been fixed correctly and there is no need to do any regression testing further.

If finds other related bugs It may be possible that the developer has fixed only the symptoms of the reported bugs without fixing the underlying bug. Moreover, there may be various ways to produce that bug. Therefore, regression tests are necessary to validate that the system does not have any related bugs.

It tests to check the effect on other parts of the program It may be possible that bug-fixing has unwanted consequences on other parts of a program. Therefore, regression tests are necessary to check the influence of changes in one part on other parts of the program.

8.5 When is Regression Testing Done?

Software Maintenance

Corrective maintenance Changes made to correct a system after a failure has been observed (usually after general release).

Adaptive maintenance Changes made to achieve continuing compatibility with the target environment or other systems.

Perfective maintenance Changes designed to improve or add capabilities.

Preventive maintenance Changes made to increase robustness, maintainability, portability, and other features.

Rapid Iterative Development

The *extreme programming* approach requires that a test be developed for each class and that this test be re-run every time the class changes.

First Step of Integration

Re-running accumulated test suites, as new components are added to successive test configurations, builds the regression suite incrementally and reveals regression bugs.

Compatibility Assessment and Benchmarking

Some test suites are designed to be run on a wide range of platforms and applications to establish conformance with a standard or to evaluate time and space performance. These test suites are meant for regression testing, but not intended to reveal regression bugs.

8.6 Regression Testing Types

Bug-Fix regression This testing is performed after a bug has been reported and fixed. Its goal is to repeat the test cases that expose the problem in the first place.

Side-Effect regression/Stability regression It involves retesting a substantial part of the product. The goal is to prove that the change has no detrimental effect on something that was earlier in order. It tests the overall integrity of the program, not the success of software fixes.

8.7 Defining Regression Test Problem

Let us first define the notations used in regression testing before defining the regression test problem.

P denotes a program or procedure,

P' denotes a modified version of P,

S denotes the specification for program P,

S' denotes the specification for program P',

$P(i)$ refer to the output of P on input i,

$P'(i)$ refer to the output of P' on input i, and

$T = \{t_1, \ldots\ldots t_n\}$ denotes a test suite or test set for P.

8.7.1 Is Regression Testing a Problem?

Regression testing is considered a problem, as the existing test suite with probable additional test cases needs to be tested again and again whenever there is a modification. The following difficulties occur in retesting:

- Large systems can take a long time to retest.
- It can be difficult and time-consuming to create the tests.
- It can be difficult and time-consuming to evaluate the tests. Sometimes, it requires a person in the loop to create and evaluate the results.
- Cost of testing can reduce resources available for software improvements.

8.7.2 Regression Testing Problem

Given a program P, its modified version P', and a test set T that was used earlier to test P; find a way to utilize T to gain sufficient confidence in the correctness of P'.

8.8 Regression Testing Techniques

There are three different techniques for regression testing. They are discussed below.

Regression test selection technique This technique attempt to reduce the time required to retest a modified program by selecting some subset of the existing test suite.

Test case prioritization technique Regression test prioritization attempts to reorder a regression test suite so that those tests with the highest priority, according to some established criteria, are executed earlier in the regression testing process rather than those with lower priority. There are two types of prioritization:

(a) **General Test Case Prioritization** For a given program P and test suite T, we prioritize the test cases in T that will be useful over a succession of subsequent modified versions of P, without any knowledge of the modified version.

(b) **Version-Specific Test Case Prioritization** We prioritize the test cases in T, when P is modified to P', with the knowledge of the changes made in P.

Test suite reduction technique It reduces testing costs by permanently eliminating redundant test cases from test suites in terms of codes or functionalities exercised.

8.8.1 Selective Retest Technique

Software maintenance includes more than 60% of development costs. In that case, testing costs dominate because many systems require labour-intensive manual testing. Selective retest techniques attempt to reduce the cost of testing by identifying the portions of P' that must be exercised by the regression test suite. Selective retesting is distinctly different from a retest-all approach that always executes every test in an existing *regression test suite* (RTS). Thus, the objective of selective retest technique is *cost reduction.* It is the process of selecting a subset of the regression test suite that tests the changes.

Following are the characteristic features of the selective retest technique:

- It minimizes the resources required to regression test a new version.
- It is achieved by minimizing the number of test cases applied to the new version.
- It is needed because a regression test suite grows with each version, resulting in broken, obsolete, uncontrollable, redundant test cases.
- It analyses the relationship between the test cases and the software elements they cover.
- It uses the information about changes to select test cases.

Rothermel and Harrold [50,51,52] have also characterized the typical steps taken by this technique in the following manner (see Fig. 8.2):

Figure 8.2 Selective Retest Technique

1. Select T' subset of T, a set of test cases to execute on P'.
2. Test P' with T', establishing correctness of P' with respect to T'.
3. If necessary, create T'', a set of new functional or structural test cases for P'.
4. Test P' with T'', establishing correctness of P' with respect to T''.
5. Create T''', a new test suite and test execution profile for P', from T, T', and T''.

Each of these steps involves the following important problems:

Regression test selection problem Step 1 involves this problem. The problem is to select a subset T' of T with which P' will be tested.

Coverage identification problem Step 3 addresses this problem. The problem is to identify portions of P' or its specifications that requires additional testing.

Test suite execution problem Steps 2 and 4 address the test suite execution problem. The problem is to execute test suites efficiently and checking test results for correctness.

Test suite maintenance problem Step 5 addresses this problem. The problem is to update and store test information.

Strategy for Test Case Selection

Effective testing requires strategies to trade-off between the two opposing needs of amplifying testing thoroughness on one side (for which a high number of test cases would be desirable) and reducing time and costs on the other (for which the fewer the test cases, the better). Given that test resources are limited, how the test cases are selected is of crucial importance. Indeed, the problem of test cases selection has been the largely dominating topic in software testing research to the extent that in the literature 'software testing' is often taken as synonymous for 'test case selection'. A decision procedure for selecting the test cases is provided by a *test criterion.*

Selection Criteria Based on Code

The motivation for code-based testing is that potential failures can only be detected if the parts of code that can cause faults are executed. All the code-based regression test selection techniques attempt to select a subset T' of T that will be helpful in establishing confidence that $P's$ functionality has been preserved where required. In this sense, all code-based test selection techniques are concerned with locating tests in T that expose faults in P'. The following tests are based on these criteria.

Fault-revealing test cases A test case t detects a fault in P' if it causes P' to fail. Hence t is fault-revealing for P'. There is no effective procedure to find the tests in T that are fault-revealing for P'. Under certain conditions, however, a regression test selection technique can select a superset of the tests in T that are fault-revealing for P'. Under these conditions, such a technique omits no tests in T that can reveal faults in P'.

Modification-revealing test cases A test case t is modification-revealing for P and P' if and only if it causes the outputs of P and P' to differ.

Modification-traversing test cases A test case t is modification-traversing if and only if it executes new or modified code in P'. These tests are useful to consider because a non-obsolete test t in T can only be modification-revealing for P and P' if it is modification-traversing for P and P'.

Regression Test Selection Techniques

A variety of regression test selection techniques have been described in the research literature. We consider the following selective retest techniques:

Minimization techniques Minimization-based regression test selection techniques attempt to select minimal sets of test cases from T that yield coverage of modified or affected portions of P. For example, the technique of Fischer *et al.* [53] uses systems of linear equations to express relationships between test cases and basic blocks (single-entry, single-exit, sequences of statements in a procedure). The technique uses a 0-1 integer programming algorithm to identify a subset T' of T that ensures that every segment that is statically reachable from a modified segment is exercised by at least one test case in T' that exercises the modified segment.

Dataflow techniques Dataflow coverage-based regression test selection techniques select test cases that exercise data interactions that have been affected by modifications. For example, the technique of Harrold and Soffa [54] requires that every definition-use pair that is deleted from P, new in P', or modified for P', should be tested. The technique selects every test case in T that, when executed on P, exercised, deleted, or modified definition-use pairs, or executed a statement containing a modified predicate.

Safe techniques Most regression test selection techniques, minimization and dataflow techniques among them, are not designed to be safe. Techniques that are not safe can fail to select a test case that would have revealed a fault in the modified program. In contrast, when an explicit set of safety conditions can be satisfied, safe regression test selection techniques guarantee that the selected subset T' contains all test cases in the original test suite T that can reveal faults in P'.

For example, the technique of Rothermel and Harrold [55] uses control-flow graph representations of P and P' and test execution profiles gathered on P, to select every test case in T that, when executed on P, exercised at least one statement that has been deleted from P, or that, when executed on P', will exercise at least one statement that is new or modified in P' (when P is executed, a statement that does not exist in P cannot be exercised).

Ad hoc/Random techniques When time constraints prohibit the use of a retest-all approach, but no test selection tool is available, developers often select test cases based on 'intuitions' or loose associations of test cases with functionality. Another simple approach is to randomly select a predetermined number of test cases from T.

Retest-all technique The retest-all technique simply reuses all existing test cases. To test P', the technique effectively selects all test cases in T.

Evaluating Regression Test Selection Techniques

To choose a technique for practical application, Rothermel *et al.* [51] have recognized the following categories for evaluating the regression test selection techniques:

Inclusiveness It measures the extent to which M chooses modification-revealing test from T for inclusion in T', where M is a regression test selection technique.

Suppose, T contains n tests that are modification-revealing for P and P', and suppose M selects m of these tests. The inclusiveness of M relative to P and P' and T is:

1. $\text{INCL}(M) = (100 \times (m/n)\%$, if $n \neq 0$
2. $\text{INCL}(M)\% = 100\%$, if $n = 0$

For example, if T contains 100 tests of which 20 are modification-revealing for P and P', and M selects 4 of these 20 tests, then M is 20% inclusive relative to P, P', and T. If T contains no modification-revealing tests, then every test selection technique is 100% inclusive relative to P, P', and T.

If for all P, P', and T, M is 100% inclusive relative to P, P', and T, then M is safe.

Precision It measures the extent to which M omits tests that are non-modification-revealing. Suppose T contains n tests that are non-modification-revealing for P and P', and suppose M omits m of these tests. The precision of M relative to P, P', and T is given by

$$\text{Precision} = 100 \times (m/n)\ \% \text{ if } n \neq 0$$

$$\text{Precision} = 100\ \% \text{ if } n = 0$$

For example, if T contains 50 tests of which 22 are non-modification-revealing for P and P', and M omits 11 of these 22 tests, then M is 50% precise relative to P, P', and T. If T contains no non-modification-revealing tests, then every test selection technique is 100% precise relative to P, P', and T.

Efficiency The efficiency of regression test selection techniques is measured in terms of their space and time requirements. Where time is concerned, a test selection technique is more economical than the retest-all technique, if the cost of selecting T' is less than the cost of running the tests in $T–T'$. Space efficiency primarily depends on the test history and program analysis information a technique must store. Thus, both space and time efficiency depends on the size of the test suite that a technique selects, and on the computational cost of that technique.

8.8.2 Regression Test Prioritization

The regression test prioritization approach is different as compared to selective retest techniques. The prioritization methods order the execution of RTS with the intention that a fault is detected earlier. In other words, regression test prioritization attempts to reorder a regression test suite so that those tests with the highest priority, according to some established criterion, are executed earlier in the regression testing process than those with a lower priority. By prioritizing the execution of a regression test suite, these methods reveal important defects in a software system earlier in the regression testing process.

This approach can be used along with the previous selective retest technique. The steps for this approach are given below:

1. Select T' from T, a set of test to execute on P'.
2. Produce T'_p, a permutation of T', such that T'_p will have a better rate of fault detection than T'.
3. Test P' with T'_p in order to establish the correctness of P' with respect to T'_p.
4. If necessary, create T'', a set of new functional or structural tests for P''.
5. Test P' with T'' in order to establish the correctness of P' with respect to T''.
6. Create T''', a new test suite for P', from T, T'_p, and T''.

Step 1 allows the tester to select T' such that it is a proper subset of T or that it actually contains every test that T contains. Furthermore, step 2 could produce T'_p so that it contains an execution ordering for the regression test that

is the same or different than the order provided by T'. The intention behind prioritizing the execution of a regression test suite is to ensure that the defects normally introduced by competent programmers are discovered earlier in the testing process.

Test prioritization will be discussed in detail in Chapter 12.

SUMMARY

This chapter discusses the importance of regression testing. It validates the changes after the code is modified or the requirements are updated. However, regression testing is considered a problem. The general strategy to validate the system is to execute all the test cases. But it is nearly impossible to test all of them due to time and budget constraints. In this case, we need to decide which parts or modules are unaffected by the changes. So the problem of regression testing lies in deciding the subset of full test cases which will validate the modifications in the software. This chapter discusses the issues of defining regression testing problem, regression testing techniques, and their evaluation.

Let us review the important concepts described in this chapter:

- Regression testing often has a strong positive influence on software quality.
- Regression number is the average number of affected test cases in the test suite that are affected by any modification to a single instruction.
- Regression testing is of two types: bug-fix regression and side-effect regression
- Regression testing is considered a problem, as the existing test suite with probable additional test cases needs to be tested again and again whenever there is a modification.
- Regression testing techniques are: regression test selection technique, test case prioritization technique, and test suite reduction technique.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. A changed version that has not passed a regression test is called _____.
   (a) baseline version
   (b) delta version
   (c) delta build
   (d) none of the above
2. Regression testability is a function of _____.
   (a) design of the program and the test suite
   (b) design of the program
   (c) test suite
   (d) none of the above

3. Regression number is computed using information about the _____.
   (a) design coverage of the program
   (b) test suite coverage of the program
   (c) number of faults
   (d) all of the above
4. Changes made to correct a system after a failure has been observed is called _____.
   (a) perfective maintenance
   (b) adaptive maintenance
   (c) corrective maintenance
   (d) all of the above
5. Changes made to achieve continuing compatibility with the target environment or other systems is called _____.
   (a) perfective maintenance
   (b) adaptive maintenance
   (c) corrective maintenance
   (d) all of the above
6. Changes designed to improve or add capabilities is called _____.
   (a) perfective maintenance
   (b) adaptive maintenance
   (c) corrective maintenance
   (d) all of the above
7. Changes made to increase robustness, maintainability, portability, and other features is called _____.
   (a) perfective maintenance
   (b) preventive maintenance
   (c) adaptive maintenance
   (d) corrective maintenance
8. Which statement best characterizes the regression test selection?
   (a) Minimize the resources required to regression test a new version
   (b) Typically achieved by minimizing the number of test cases applied to the new version
   (c) Select a test case which has caused the problem
   (d) None of the above
9. Problem to select a subset T' of T with which P' will be tested is called _____.
   (a) coverage identification problem
   (b) test suite execution problem
   (c) regression test selection problem
   (d) test suite maintenance problem

10. The problem to identify portions of *P'* or its specification that requires additional testing is called _____.
    (a) coverage identification problem
    (b) test suite execution problem
    (c) regression test selection problem
    (d) test suite maintenance problem
11. The problem to execute test suites efficiently and checking test results for correctness is called _____.
    (a) coverage identification problem
    (b) test suite execution problem
    (c) regression test selection problem
    (d) test suite maintenance problem
12. The problem to update and store test information is called _____.
    (a) coverage identification problem
    (b) test suite execution problem
    (c) regression test selection problem
    (d) test suite maintenance problem
13. A test case t is modification-revealing for *P* and *P'* if _____.
    (a) it executes new or modified code in *P'*
    (b) it detect a fault in *P'* if it causes *P'* to fail
    (c) it causes the outputs of *P* and *P'* to differ
    (d) all of the above
14. A test case *t* is modification-traversing if _____.
    (a) it executes new or modified code in *P'*
    (b) it detect a fault in *P'* if it causes *P'* to fail
    (c) it causes the outputs of *P* and *P'* to differ
    (d) all of the above
15. A test case *t* is fault-revealing if _____.
    (a) it executes new or modified code in *P'*
    (b) it detect a fault in *P'* if it causes *P'* to fail
    (c) it causes the outputs of *P* and *P'* to differ
    (d) all of the above
16. Regression testing is helpful in _____.
    (a) detecting bugs
    (b) detecting undesirable side effects by changing the operating environment
    (c) integration testing
    (d) all of the above

Review Questions

1. When is a system said to regress?
2. What is the difference between baseline version and delta version?
3. How do you classify a program as regression testable?
4. What is regression number?
5. How does regression testing help in producing a quality software?
6. What are the situations to perform regression testing?
7. Why is regression testing a problem?
8. Define regression testing problem with an example.
9. What is the aim of selective retest technique?
10. Define the following:
    (a) Regression test selection problem
    (b) Coverage identification problem
    (c) Test suite execution problem
    (d) Test suite maintenance problem
11. What is regression test prioritization test technique?
12. What is the difference between fault-revealing test cases, modification-revealing test cases, and modification-traversing test cases?
13. Explain the following regression test selection techniques:
    (a) Minimization
    (b) Dataflow
    (c) Safe
    (d) Ad hoc
14. What are the different parameters for evaluating regression test selection techniques?
15. *T* contains 90 tests of which 20 are modification-revealing for *P* and *P'* and *M* selects 12 of these 20 tests, then calculate the inclusiveness of *M* relative to *P*, *P'*, and *T*.
16. *T* contains 100 tests of which 64 are non-modification-revealing for *P* and *P'* and *M* omits 43 of these 64 tests, then calculate the precision of *M* relative to *P*, *P'*, and *T*.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Part 3 Managing the Test Process

CHAPTERS

Chapter 9:
Test Management

Chapter 10:
Software Metrics

Chapter 11:
Testing Metrics for Monitoring and Controlling the Testing Process

Chapter 12:
Efficient Test Suite Management

Testing process, still in its infancy, needs to be managed from various perspectives. In many industries, no test organization hierarchy is followed. Rather, there is a single development team performing every activity including testing. To perform effective testing, we need a testing hierarchy of people with various roles. To manage a testing process, thus, the first step is to get a test organization in place. Next, we must have a master test plan which guides us when and how to perform various testing activities.

In this part, we discuss various test plans including verification and validation test plans. Various specifications such as test design, test case, test result specifications, have been explained in order to carry out systematic documented testing.

To manage the testing process, an organization is not sufficient; we also need to monitor and control various testing activities in order to meet the testing and project milestones. But we cannot monitor and control the test activities unless we have some methods of measurements.

This part discusses software and testing metrics. These metrics help in understanding, monitoring, and controlling various testing activities.

There is one major problem while managing the test process. The test suite prepared to test the system and to perform regression testing is too large to test. We are not able to execute all the test cases. In such a situation, if we leave out some of the test cases, we are unsure if the left-out test cases are critical test cases to be executed or not. This part discusses various techniques adopted to reduce the number of test cases in order to perform effective test cases and manage the test suite.

This part will make ground for the following concepts:

- Structure of test organization
- Test planning
- Test design and report specifications
- Testing metrics
- Test suite reduction methods

Chapter

9

Test Management

Test management is concerned with both test resource and test environment management. It is the role of test management to ensure that new or modified service products meet the business requirements for which they have been developed or enhanced. The key elements of test management are discussed below.

OBJECTIVES

After reading this chapter, you should be able to understand:

- Elements of test management
- Capabilities of a tester
- Structure of testing group
- Test plan components
- Master test plan
- Verification test plan
- Unit test plan
- Integration test plan
- Function test plan
- System test plan
- Acceptance test plan
- Test design specifications
- Test case specifications
- Test result specifications: test log, test incident report, and test summary report

Test organization It is the process of setting up and managing a suitable test organizational structure and defining explicit roles. The project framework under which the testing activities will be carried out is reviewed, high-level test phase plans are prepared, and resource schedules are considered. Test organization also involves the determination of configuration standards and defining the test environment.

Test planning The requirements definition and design specifications facilitate the identification of major test items and these may necessitate updating the test strategy. A detailed test plan and schedule is prepared with key test responsibilities being indicated.

Detailed test design and test specifications A detailed design is the process of designing a meaningful and useful structure for the tests as a whole. It specifies the details of the test approach for a software functionality or feature and identifying the associated test cases.

Test monitoring and assessment It is the ongoing monitoring and assessment to check the integrity of development and construction. The status of configuration items should be reviewed against the phase plans and the test progress reports prepared, the ensure the verification and validation activities are correct.

Product quality assurance The decision to negotiate the acceptance testing program and the release and commissioning of the service product is subject to the 'product assurance' role being satisfied with the outcome of the verification and validation activities. Product assurance may oversee some of the test activity and may participate in process reviews.

9.1 Test Organization

Since testing is viewed as a process, it must have an organization such that a testing group works for better testing and high quality software. The testing group is responsible for the following activities:

- Maintenance and application of test policies
- Development and application of testing standards
- Participation in requirement, design, and code reviews
- Test planning
- Test execution
- Test measurement
- Test monitoring
- Defect tracking
- Acquisition of testing tools
- Test reporting

The staff members of such a testing group are called *test specialists* or *test engineers* or simply *testers*. A tester is not a developer or an analyst. He does not debug the code or repair it. He is responsible for ensuring that testing is effective and quality issues are being addressed. The skills a tester should possess are discussed below.

Personal and Managerial Skills

- Testers must be able to contribute in policy-making and planning the testing activities.
- Testers must be able to work in a team.
- Testers must be able to organize and monitor information, tasks, and people.
- Testers must be able to interact with other engineering professionals, software quality assurance staff, and clients.
- Testers should be capable of training and mentoring new testers.

- Testers should be creative, imaginative, and experiment-oriented.
- Testers must have written and oral communication skills.

Technical Skills

- Testers must be technically sound, capable of understanding software engineering principles and practices.
- Testers must be good in programming skills.
- Testers must have an understanding of testing basics, principles, and practices.
- Testers must have a good understanding and practice of testing strategies and methods to develop test cases.
- Testers must have the ability to plan, design, and execute test cases with the goal of logic coverage.
- Testers must have technical knowledge of networks, databases, operating systems, etc. needed to work in a the project environment.
- Testers must have the knowledge of configuration management.
- Testers must have the knowledge of testware and the role of each document in the testing process.
- Testers must have know about quality issues and standards.

9.2 Structure of Testing Group

Testing is an important part of any software project. One or two testers are not sufficient to perform testing, especially if the project is too complex and large. Therefore, many testers are required at various levels. Figure 9.1 shows different types of testers in a hierarchy.

Test Manager A test manager occupies the top level in the hierarchy. He has the following responsibilities:

(i) He is the key person in the testing group who will interact with project management, quality assurance, and marketing staff.
(ii) Takes responsibility for making test strategies with detailed master planning and schedule.
(iii) Interacts with customers regarding quality issues.
(iv) Acquires all the testing resources including tools.
(v) Monitors the progress of testing and controls the events.
(vi) Participates in all static verification meetings.
(vii) Hires, fires, and evaluates the test team members.

Figure 9.1 Testing Group Hierarchy

Test Leader The next tester in the hierarchy is the test leader who assists the test manager in meeting testing and quality goals. The prime responsibility of a test leader is to lead a team of test engineers who are working at the leaf-level of the hierarchy. The following are his responsibilities:

(i) Planning the testing tasks given by the test manager.
(ii) Assigning testing tasks to test engineers who are working under him.
(iii) Supervising test engineers.
(iv) Helping the test engineers in test case design, execution, and reporting.
(v) Providing tool training, if required.
(vi) Interacting with customers.

Test Engineers Test engineers are highly experienced testers. They work under the leadership of the test leader. They are responsible for the following tasks:

(i) Designing test cases.
(ii) Developing test harness.
(iii) Set-up test laboratories and environment.
(iv) Maintain the test and defect repositories.

Junior Test Engineers Junior test engineers are newly hired testers. They usually are trained about the test strategy, test process, and testing tools. They participate in test design and execution with experienced test engineers.

9.3 Test Planning

There is a general human tendency to 'get on with the next thing', especially under pressure [1]. This is true for the testers who are always short of time.

However, if resources are to be utilized intelligently and efficiently during the earlier testing phases and later phases, these are repaid many times over. The time spent on planning the testing activities early is never wasted and usually the total time cycle is significantly shorter.

According to the test process as discussed in STLC, testing also needs planning as is needed in SDLC. Since software projects become uncontrolled if not planned properly, the testing process is also not effective if not planned earlier. Moreover, if testing is not effective in a software project, it also affects the final software product. Therefore, for a quality software, testing activities must be planned as soon as the project planning starts.

A *test plan* is defined as a document that describes the scope, approach, resources, and schedule of intended testing activities. Test plan is driven with the business goals of the product. In order to meet a set of goals, the test plan identifies the following:

- Test items
- Features to be tested
- Testing tasks
- Tools selection
- Time and effort estimate
- Who will do each task
- Any risks
- Milestones

9.3.1 Test Plan Components

IEEE Std 829–1983 [56] has described the test plan components. These components (see Fig. 9.2) must appear for every test item. These components are described below.

Test Plan Identifier

Each test plan is tagged with a unique identifier so that it is associated with a project.

Introduction

The test planner gives an overall description of the project with:

- Summary of the items and features to be tested.
- The requirement and the history of each item (optional).
- High-level description of testing goals.

- References to related documents, such as project authorization, project plan, QA plan, configuration management plan, relevant policies, and relevant standards.

Test Plan Components

- Test Plan Identifier
- Introduction
- Test-Item to be tested
- Features to be tested
- Features not to be tested
- Approach
- Item Pass/Fail Criteria
- Suspension criteria and resumption requirements
- Test deliverables
- Testing tasks
- Environmental needs
- Responsibilities
- Staffing and training needs
- Scheduling
- Risks and contingencies
- Testing costs
- Approvals

Figure 9.2 Test plan components

Test-Item to be Tested

- Name, identifier, and version of test items.
- Characteristics of their transmitting media where the items are stored, for example, disk, CD, etc.
- References to related documents, such as requirements specification, design specification, users' guide, operations guide, installation guide.
- References to bug reports related to test items.
- Items which are specifically not going to be tested (optional).

Features to be Tested

This is a list of what needs to be tested from the user's viewpoint. The features may be interpreted in terms of functional and quality requirements.

- All software features and combinations of features are to be tested.
- References to test-design specifications associated with each feature and combination of features.

Features Not to be Tested

This is a list of what should 'not' be tested from both the user's viewpoint and the configuration management/version control view:

- All the features and the significant combinations of features which will not be tested.
- Identify *why* the feature is not to be tested. There can be many reasons:
  - (i) Not to be included in this release of the software.
  - (ii) Low-risk, has been used before, and was considered stable.
  - (iii) Will be released but not tested or documented as a functional part of the release of this version of the software.

Approach

- Overall approach to testing.
- For each major group of features or combinations of features, specify the approach.
- Specify major activities, techniques, and tools which are to be used to test the groups.
- Specify the metrics to be collected.
- Specify the number of configurations to be tested.
- Specify a minimum degree of comprehensiveness required.
- Identify which techniques will be used to judge comprehensiveness.
- Specify any additional completion criteria.
- Specify techniques which are to be used to trace requirements.
- Identify significant constraints on testing, such as test-item availability, testing-resource availability, and deadline.

Item Pass/Fail Criteria

This component defines a set of criteria based on which a test case is passed or failed. The failure criteria is based on the severity levels of the defect. Thus, an acceptable severity level for the failures revealed by each test case is specified and used by the tester. If the severity level is beyond an acceptable limit, the software fails.

Suspension Criteria and Resumption Requirements

Suspension criteria specify the criteria to be used to suspend all or a portion of the testing activities, while resumption criteria specify when the testing can resume after it has been suspended.

For example, system integration testing in the integration environment can be suspended in the following circumstances:

- Unavailability of external dependent systems during execution.
- When a tester submits a 'critical' or 'major' defect, the testing team will call for a break in testing while an impact assessment is done.

System integration testing in the integration environment may be resumed under the following circumstances:

- When the 'critical' or 'major' defect is resolved.
- When a fix is successfully implemented and the testing team is notified to continue testing.

Test Deliverables

- Identify deliverable documents: test plan, test design specifications, test case specifications, test item transmittal reports, test logs, test incident reports, test summary reports, and test harness (stubs and drivers).
- Identify test input and output data.

Testing Tasks

- Identify the tasks necessary to prepare for and perform testing.
- Identify all the task interdependencies.
- Identify any special skills required.

All testing-related tasks and their interdependencies can be shown through a *work breakdown structure* (WBS). WBS is a hierarchical or tree-like representation of all testing tasks that need to be completed in a project.

Environmental Needs

- Specify necessary and desired properties of the test environment: physical characteristics of the facilities including hardware, communications and system software, the mode of usage (i.e. stand-alone), and any other software or supplies needed.
- Specify the level of security required.
- Identify any special test tools needed.
- Identify any other testing needs.
- Identify the source for all needs which are not currently available.

Responsibilities

- Identify the groups responsible for managing, designing, preparing, executing, checking, and resolving.

- Identify the groups responsible for providing the test items identified in the test items section.
- Identify the groups responsible for providing the environmental needs identified in the environmental needs section.

Staffing and Training Needs

- Specify staffing needs by skill level.
- Identify training options for providing necessary skills.

Scheduling

- Specify test milestones.
- Specify all item transmittal events.
- Estimate the time required to perform each testing task.
- Schedule all testing tasks and test milestones.
- For each testing resource, specify a period of use.

Risks and Contingencies

Specify the following overall risks to the project with an emphasis on the testing process:

- Lack of personnel resources when testing is to begin.
- Lack of availability of required hardware, software, data, or tools.
- Late delivery of the software, hardware, or tools.
- Delays in training on the application and/or tools.
- Changes to the original requirements or designs.
- Complexities involved in testing the applications.

Specify the actions to be taken for various events. An example is given below.

Requirements definition will be complete by January 1, 20XX and, if the requirements change after that date, the following actions will be taken:

The test schedule and the development schedule will move out an appropriate number of days. This rarely occurs, as most projects tend to have fixed delivery dates.

- The number of tests performed will be reduced.
- The number of acceptable defects will increase.
- These two items may lower the overall quality of the delivered product.
- Resources will be added to the test team.
- The test team will work overtime.

- The scope of the plan may be changed.
- There may be some optimization of resources. This should be avoided, if possible, for obvious reasons.

The management team is usually reluctant to accept scenarios such as the one mentioned above even though they have seen it happen in the past. The important thing to remember is that, if you do nothing at all, testing is cut back or omitted completely, neither of which should be an acceptable option.

Testing Costs

The IEEE standard has not included this component in its specification. However, it is a usual component of any test plan, as test costs are allocated in the total project plan. To estimate the costs, testers will need tools and techniques. The following is a list of costs to be included:

- Cost of planning and designing the tests
- Cost of acquiring the hardware and software required for the tests
- Cost to support the environment
- Cost of executing the tests
- Cost of recording and analysing the test results
- Cost of training the testers, if any
- Cost of maintaining the test database

Approvals

- Specify the names and titles of all the people who must approve the plan.
- Provide space for signatures and dates.

9.3.2 Test Plan Hierarchy

Test plans can be organized in several ways depending on the organizational policy. There is often a hierarchy of plans that includes several levels of quality assurance and test plans. At the top of the plan hierarchy is a master plan which gives an overview of all verification and validation activities for the project, as well as details related to other quality issues such as audits, standards, and configuration control. Below the master test plan, there is individual planning for each activity in verification and validation, as shown in Fig. 9.3.

The test plan at each level of testing must account for information that is specific to that level, e.g. risks and assumptions, resource requirements, schedule, testing strategy, and required skills. The test plans according to each level are written in an order such that the plan prepared first is executed last,

i.e. the acceptance test plan is the first plan to be formalized but is executed last. The reason for its early preparation is that the things required for its completion are available first.

Figure 9.3 Test Plan Hierarchy

9.3.3 Master Test Plan

The master test plan provides the highest level description of verification and validation efforts and drives the testing at various levels. General project information is used to develop the master test plan.

The following topics must be addressed before planning:

- Project identification
- Plan goals
- Summary of verification and validation efforts
- Responsibilities conveyed with the plan
- Software to be verified and validated
- Identification of changes to organization standards

Verification and validation planning may be broken down into the following steps:

- Identify the V&V scope
- Establish specific objectives from the general project scope
- Analyse the project input prior to selecting V&V tools and techniques
- Select tools and techniques
- Develop the software verification and validation plan (SVVP).

Discussed below are the major activities of V&V planning.

Master Schedule

- The master schedule summarizes various V&V tasks and their relationship to the overall project.
- Describes the project life cycle and project milestones including completion dates.
- Summarizes the schedule of V&V tasks and how verification and validation results provide feedback to the development process to support overall project management functions.
- Defines an orderly flow of material between project activities and V&V tasks. Use reference to PERT, CPM, and Gantt charts to define the relationship between various activities.

Resource Summary

This activity summarizes the resources needed to perform V&V tasks, including staffing, facilities, tools, finances, and special procedural requirements such as security, access rights, and documentation control. In this activity,

- Use graphs and tables to present resource utilization.
- Include equipment and laboratory resources required.
- Summarize the purpose and cost of hardware and software tools to be employed.
- Take all resources into account and allow for additional time and money to cope with contingencies.

Responsibilities

Identify the organization responsible for performing V&V tasks. There are two levels of responsibilities—general responsibilities assigned to different organizations and specific responsibilities for the V&V tasks to be performed, assigned to individuals. General responsibilities should be supplemented with specific responsibility for each task in the V&V plan.

Tools, Techniques, and Methodology

Identify the special software tools, techniques, and methodologies to be employed by the V&V team. The purpose of each should be defined and plans for the acquisition, training, support, and qualification of each should be described. This section may be in a narrative or graphic format. A separate tool plan may be developed for software tool acquisition, development, or modification. In this case, a separate tool plan section may be added to the plan.

9.3.4 Verification Test Plan

Verification test planning includes the following tasks:

- The item on which verification is to be performed.
- The method to be used for verification: review, inspection, walkthrough.
- The specific areas of the work product that will be verified.
- The specific areas of the work product that will not be verified.
- Risks associated.
- Prioritizing the areas of work product to be verified.
- Resources, schedule, facilities, tools, and responsibilities.

9.3.5 Validation Test Plan

Validation test planning includes the following tasks:

- Testing techniques
- Testing tools
- Support software and documents
- Configuration management
- Risks associated, such as budget, resources, schedule, and training

Unit Test Plan

Generally, unit tests are performed by the module developer informally. In this case, bugs are not recorded and do not become a part of the unit's history. It is a bad practice because these bugs' history will not be re-used, which otherwise becomes important especially in real-time systems. Therefore, to implement unit testing formally and to make it an effective testing for other projects, it is necessary to plan for unit test. To prepare for unit test, the developer must do the following tasks:

- Prepare the unit test plan in a document form.
- Design the test cases.
- Define the relationship between tests.
- Prepare the test harness, i.e. stubs and drivers, for unit testing.

Discussed below are the tasks contained in a test plan document.

Module overview Briefly define the purpose of this module. This may require only a single phrase, i.e. calculates overtime pay amount, calculates equipment depreciation, performs date edit validation, or determines sick pay eligibility, etc.

Inputs to module Provide a brief description of the inputs to the module under test.

Outputs from module Provide a brief description of the outputs from the module under test.

Logic flow diagram Provide a logic flow diagram, if additional clarity is required. For example, DFD, STD, and ERD.

Test approach Decide the method to test the module.

Features to be tested Identify the features to be tested for the module.

Features not to be tested Identify the features not be tested due to some constraints. Also, assess the risks of not testing these features.

Test constraints Indicate anticipated limitations on the test due to test conditions, such as interfaces, equipment, personnel, and databases.

Test harness Describe the requirements for test harness that interfaces with the units to be tested.

Interface modules Identify the modules that interface with this module indicating the nature of the interface: outputs data to, receives input data from, internal

program interface, external program interface, etc. Identify the sequencing required for subsequent string tests or sub-component integration tests.

Test tools Identify any tools employed to conduct unit testing. Specify any stubs or utility programs developed or used to invoke tests. Identify names and locations of these aids for future regression testing. If data is supplied from the unit test of a coupled module, specify module relationship.

Archive plan Specify how and where the data is archived for use in subsequent unit tests. Define any procedures required to obtain access to data or tools used in the testing effort. The unit test plans are normally archived with the corresponding module specifications.

Updates Define how updates to the plan will be identified. Updates may be required due to enhancements, requirement changes, etc. The same unit test plan should be re-used with revised or appended test cases identified in the update section.

Milestones List the milestone events and dates for unit testing.

Budget List the funds allocated to test this module. A checklist can also be maintained, as shown below:

- Does the test plan completely test the functionality of the unit as defined in the design specification and in functional specification?
- Are the variable (including global variables) inputs to the unit included in the features to be tested in the test plan?
- Are the variable outputs from the unit result included in the features to be tested?
- Is every procedure in the design exercised by the test?
- Does the test plan include critical procedures to demonstrate that performance meets stated requirements?
- For units that interface with other units, are the invalid features tested to test for defensive coding?

Integration Test Plan

An integration test plan entails the development of a document, which instructs a tester what tests to perform in order to integrate and test the already existing individual code modules. The objective of the integration test plan activity is to develop a plan which specifies the necessary steps needed to integrate individual modules and test the integration. It also helps the technical team to think through the logical sequence of integration activities, so

that their individual detailed development plans are well-synchronized, and integration happens in a reasonable manner.

Create a draft of the plan very early in the execution phase. By this time, you will know from the high-level design work in the initiation phase, what major pieces of hardware and/or software will have to be integrated. An early look at integration sequencing will help you sanity-check the amount of time you have put into the schedule for integration testing.

Discussed below are the tasks to be performed under integration test plan.

Documents to be used All the documents needed should be arranged, e.g. SRS, SDD, user manual, usage scenarios containing DFDs, data dictionaries, state charts, etc.

Modules for integration It should be planned which modules will be integrated and tested. There can be many criteria for choosing the modules for integrating. Some are discussed below.

- **Availability** All the modules may not be ready at one time. But we cannot delay the integration and testing of modules which are ready. Therefore, the modules which are ready for integration must be integrated and tested first.
- **Subsystems** The goal of integration testing is also to have some working subsystems and combine them into the working system as a whole. While planning for integration, the subsystems are selected based on the requirements, user needs, and availability of the modules. The subsystems may also be prioritized based on key or critical features. Sometimes, developers need to show the clients some subsystems that are working at the time of integration testing.

Strategy for integration The strategy selection for integration testing should also be well-planned. Generally, sandwich integration testing is adopted.

Test harness A factor to be considered while developing an integration test plan is the amount of 'dead code' or test harness that will have to be developed. The *drivers* and *stub* code that is developed is thrown away and never used again, so a minimum amount of development of this code is desired. Thus, if the program uses a lot of low-level routines such that it is not practical to write a driver for every low-level routine, then perhaps the level above should be developed first, with drivers for it.

List of tasks to be tested The functionalities and interfaces to be tested through the integration of modules must also be planned.

List of tasks not to be tested Plan the functionalities and interfaces not to be tested due to whatever reasons.

Function Test Plan

During planning, the test lead with assistance from the test team defines the scope, schedule, and deliverables for the function test cycle. The test lead delivers a test plan (document) and a test schedule (work plan)—these often undergo several revisions during the testing cycle. The following are the tasks under function test planning:

- **List the objectives of function testing** A list of the overall objectives for performing function testing is prepared.
- **Partitioning/Functional decomposition** Functional decomposition of a system (or partitioning) is the breakdown of a system into its functional components or functional areas. Another group in the organization may take responsibility for the functional decomposition (or model) of the system, but the testing organization should still review this deliverable for completeness before accepting it into the test organization. If the functional decomposition or partitions have not been defined or are deemed insufficient, then the testing organization will have to take responsibility for creating and maintaining the partitions.
- **Traceability matrix formation** Test cases need to be traced/mapped back to the appropriate requirement. A function coverage matrix is prepared. This matrix is a table, listing specific functions to be tested, the priority for testing each function, and test cases that contain tests for each function. Once all the aspects of a function have been tested by one or more test cases, the test design activity for that function can be considered complete. This approach gives a more accurate picture of the application when coverage analysis is done.
- **List the functions to be tested** A list of all the functions mentioned in the traceability matrix with their appropriate details needed for planning, is prepared.

System Test Plan

One of the most important parts of software development is testing. Before you implement a new software into a system, you must be sure it won't crash and that proper bug checking has been done. Testing ensures that your system runs properly and efficiently and meets all the requirements. To fully ensure that your system is free of defects, testing should follow a system test plan and must be thorough and complete.

Often people rush to implement a new program and then spend hours repairing the system, losing many hours of productivity. The experts prevent this by using a system test plan. Basically, a system test plan is a systematic approach to testing a system. The plan contains a detailed understanding of what the eventual workflow will be. This will help keep the project on schedule and make the integration of the new system as smooth as possible. We should not build a house without blueprints, nor should we run a new system without having created a proper system test plan.

System testing is also difficult, as there are no design methodologies for test cases because requirements and objectives do not describe the functions precisely. However, requirements are general. Therefore, a system test plan first requires that requirements are formulated specifically. For this system, test cases are divided into some categories, according to which the system test plan is described. For example, system testing is described in terms of performance testing, then the test plans are prepared according to performance checking. Similarly, test plans can be prepared for stress testing, compatibility testing, security testing, etc.

The following steps describe how a system test plan is implemented.

Partition the requirements Partition the requirements into logical categories and for each category, develop a list of detailed requirements and plan.

System description Provide a chart and briefly describe the inputs, outputs, and functions of the software being tested as a frame of reference for the test descriptions.

Features to be tested The set of functionalities of the test items to be tested must be recognized, as there may be several criteria for system testing.

Strategy and reporting format The strategy to be followed in system testing should also be planned. The type of tests to be performed, standards for documentation, mechanism for reporting, and special considerations for the type of project must also be planned, e.g. a test strategy might say that black-box testing techniques will be used for functional testing.

Develop a requirement coverage matrix Develop a requirement coverage matrix which is a table in which an entry describes a specific subtest, priority of the subtest, the specific test cases in which the subtest appears. This matrix specifies the functions that are to be tested, defining the objectives of the test.

Smoke test set The system test plan also includes a group of test cases that establish that the system is stable and all major functionalities are working. This group of test cases is referred to as *smoke tests* or *testability assessment criteria.*

Entry/Exit criteria The entry criteria will contain some exit criteria from the previous level as well as establishment of test environment, installation of tools, gathering of data, and recruitment of testers, if necessary. The exit criteria also need to be established, e.g. all test cases have been executed and no major identified bugs are open.

Suspension criteria It identifies the conditions that suspend testing, e.g. critical bugs preventing further testing.

Resource requirements State the resource requirements including documents, equipment, software, and personnel.

Test team State the members who are in the test team and enlist their assignments.

Participating organizations System testing may not be done by a single organization, as the software to be developed may be large with many modules being developed by various organizations. Thus, integrated system testing needs people from all these organizations. Identify the participating organizations and fix the date and time with them.

Extent and constraints Since the system testing of a large software system needs many resources, some might not be available. Therefore, indicate the extent of testing, e.g. total or partial. Include the rationale for the partial one. Similarly, indicate anticipated limitations on the test due to test conditions, such as interfaces, equipment, personnel, and database.

Schedule estimation The time schedule for accomplishing testing milestones should also be planned. But this plan should be in tune with the time allocation in the project plan for testing.

Acceptance Test Plan

It is important to have the acceptance criteria defined so that acceptance testing is performed against those criteria. It means that in the acceptance test plan, we must have all the acceptance criteria defined in one document. If these are not available, then prepare them and plan the acceptance test accordingly. Acceptance criteria are broadly defined for functionality requirements, performance requirements, interface quality requirements, and overall software quality requirements.

Another point in acceptance testing plan is to decide the criticality of acceptance features defined. It is necessary to define the criticality. If the system fails the high critical acceptance requirement, then it should not pass the acceptance testing.

9.4 Detailed Test Design and Test specifications

The ultimate goal of test management is to get the test cases executed. Till now, test planning has not provided the test cases to be executed. Detailed test designing for each validation activity maps the requirements or features to the actual test cases to be executed. One way to map the features to their test cases is to analyse the following:

- Requirement traceability
- Design traceability
- Code traceability

The analyses can be maintained in the form of a traceability matrix (see Table 9.1) such that every requirement or feature is mapped to a function in the functional design. This function is then mapped to a module (internal design and code) in which the function is being implemented. This in turn is linked to the test case to be executed. This matrix helps in ensuring that all requirements have been covered and tested. Priority can also be set in this table to prioritize the test cases.

Table 9.1 Traceability matrix

Requirement/Feature	Functional Design	Internal Design/Code	Test cases
R1	F1, F4,F5	abc.cpp, abc.h	T5, T8,T12,T14

9.4.1 Test Design Specification

A test design specification should have the following components according to IEEE recommendation [56]:

Identifier A unique identifier is assigned to each test design specification with a reference to its associated test plan.

Features to be tested The features or requirements to be tested are listed with reference to the items mentioned in SRS/SDD.

Approach refinements In the test plan, an approach to overall testing was discussed. Here, further details are added to the approach. For example, special testing techniques to be used for generating test cases are explained.

Test case identification The test cases to be executed for a particular feature/function are identified here, as shown in Table 9.1. Moreover, test procedures are also identified. The test cases contain input/output information and the test procedures contain the necessary steps to execute the tests. Each test design specification is associated with test cases and test procedures. A test case may be associated with more than one test design specifications.

Feature pass/fail criteria The criteria for determining whether the test for a feature has passed or failed, is described.

9.4.2 Test Case Specifications

Since the test design specifications have recognized the test cases to be executed, there is a need to define the test cases with complete specifications. The test case specification document provides the actual values for input with expected outputs. One test case can be used for many design specifications and may be re-used in other situations. A test case specification should have the following components according to IEEE recommendation [56]:

Test case specification identifier A unique identifier is assigned to each test case specification with a reference to its associated test plan.

Purpose The purpose of designing and executing the test case should be mentioned here. It refers to the functionality you want to check with this test case.

Test items needed List the references to related documents that describe the items and features, e.g. SRS, SDD, and user manual.

Special environmental needs In this component, any special requirement in the form of hardware or software is recognized. Any requirement of tool may also be specified.

Special procedural requirements Describe any special condition or constraint to run the test case, if any.

Inter-case dependencies There may be a situation that some test cases are dependent on each other. Therefore, previous test cases which should be run prior to the current test case must be specified.

Input specifications This component specifies the actual inputs to be given while executing a test case. The important thing while specifying the input values is not to generalize the values, rather specific values should be provided. For example, if the input is in angle, then the angle should not be specified as a range between 0 and 360, but a specific value like 233 should be specified. If there is any relationship between two or more input values, it should also be specified.

Test procedure The step-wise procedure for executing the test case is described here.

Output specifications Whether a test case is successful or not is decided after comparing the output specifications with the actual outputs achieved. Therefore, the output should be mentioned complete in all respects. As in the case of input specifications, output specifications should also be provided in specific values.

Example 9.1

There is a system for railway reservation system. There are many functionalities in the system, as given below:

S. No.	Functionality	Function ID in SRS	Test cases
1	Login the system	F3.4	T1
2	View reservation status	F3.5	T2
3	View train schedule	F3.6	T3
4	Reserve seat	F3.7	T4
5	Cancel seat	F3.8	T5
6	Exit the system	F3.9	T6

Suppose we want to check the functionality corresponding to 'view reservation status'. Its test specification is given in Fig. 9.4.

- **Test case Specification Identifier**
  T2
- **Purpose**
  To check the functionality of 'View Reservation Status'
- **Test Items Needed**
  Refer function F3.5 in SRS of the system.
- **Special Environmental Requirements**
  Internet should be in working condition. Database software through which the data will be retrieved should be in running condition.
- **Special Procedural Requirements**
  The function 'Login' must be executed prior to the current test case.
- **Inter-case Dependencies**
  T1 test case must be executed prior to the current test case execution.
- **Input Specifications**
  Enter PNR number in 10 digits between 0–9 as given below:
  4102645876
  21A2345672
  234
  asdgggggggg
- **Test Procedure**
  Press 'View Reservation status' button.
  Enter PNR number and press ENTER.
- **Output Specifications**
  The reservation status against the entered PNR number is displayed as S12 or RAC13 or WL123 as applicable.

Figure 9.4 Test specifications

9.4.3 Test Procedure Specifications

A test procedure is a sequence of steps required to carry out a test case or a specific task. This can be a separate document or merged with a test case specification.

9.4.4 Test Result Specifications

There is a need to record and report the testing events during or after the test execution, as shown in Fig. 9.5.

Figure 9.5 Test result specifications

Test Log

Test log is a record of the testing events that take place during the test. Test log is helpful for bug repair or regression testing. The developer gets valuable clues from this test log, as it provides snapshots of failures. The format for preparing a test log according to IEEE [56] is given below:

- **Test log identifier**
- **Description** Mention the items to be tested, their version number, and the environment in which testing is being performed.
- **Activity and event entries** Mention the following:
  - (i) Date
  - (ii) Author of the test
  - (iii) Test case ID
  - (iv) Name the personnel involved in testing

(v) For each execution, record the results and mention pass/fail status

(vi) Report any anomalous unexpected event, before or after the execution

The test log corresponding to Example 9.1 is given in Fig. 9.6.

- **Test Log Identifier**
  TL2
- **Description**
  Function 3.5 in SRS v 2.1. The function tested in Online environment with Internet.
- **Activity and Event Entries**
  Mention the following:
  (i) Date: 22/03/2009
  (ii) Author of test: XYZ
  (iii) Test case ID: T2
  (iv) Name of the personnel involved in testing: ABC, XYZ
  (v) For each execution, record the results and mention pass/fail status
  The function was tested with the following inputs:

Inputs	Results	Status
4102645876	S12	Pass
21A2345672	S14	Fail
234	Enter correct 10 digit PNR number	Pass
asdgggggggg	RAC12	Fail

  (vi) Report any anomalous unexpected event before or after the execution.
  Nil

Figure 9.6 Sample test log for Example 9.1

Test Incident Report

This is a form of bug report. It is the report about any unexpected event during testing which needs further investigation to resolve the bug. Therefore, this report completely describes the execution of the event. It not only reports the problem that has occurred but also compares the expected output with the actual results. Listed below are the components of a test incident report [56]:

- **Test incident report identifier**
- **Summary** Identify the test items involved, test cases/procedures, and the test log associated with this test.
- **Incident description** It describes the following:
  (i) Date and time
  (ii) Testing personnel names

(iii) Environment
(iv) Testing inputs
(v) Expected outputs
(vi) Actual outputs
(vii) Anomalies detected during the test
(viii) Attempts to repeat the same test

- **Impact** The originator of this report will assign a severity value/rank to this incident so that the developer may know the impact of this problem and debug the critical problem first.

The test log corresponding to Example 9.1 is given in Fig. 9.7.

- **Test Incident Report Identifier**
  TI2
- **Summary**
  Function 3.5 in SRS v 2.1. Test Case T2 and Test Log TL2.
- **Incident Description**
  It describes the following:
  (i) Date and time: 23/03/2009, 2.00pm
  (ii) Testing personnel names: ABC, XYZ
  (iii) Environment: Online environment with X database
  (iv) Testing inputs
  (v) Expected outputs
  (vi) Actual outputs
  (vii) Anomalies detected during the test
  (viii) Attempts to repeat the same test

Testing Inputs	Expected outputs	Actual outputs	Anomalies detected	Attempts to repeat the same test
4102645876	S12	S12	nil	–
21A2345672	Enter correct 10 digit PNR number	S14	Alphabet is being accepted in the input.	3
234	Enter correct 10 digit PNR number	Enter correct 10 digit PNR number	nil	–
asdgggggggg	Enter correct 10 digit PNR number	RAC12	Alphabet is being accepted in the input.	5

- **Impact**
  The severity value is 1(Highest).

Figure 9.7 Sample test incident report for Example 9.1

Test Summary Report

It is basically an evaluation report prepared when the testing is over. It is the summary of all the tests executed for a specific test design specification. It can provide the measurement of how much testing efforts have been applied for the test. It also becomes a historical database for future projects, as it provides information about the particular type of bugs observed.

The test summary report corresponding to Example 9.1 is given in Fig. 9.8.

- **Test Summary Report Identifier**
  TS2
- **Description**
  SRS v 2.1

S. No.	Functionality	Function ID in SRS	Test cases
1	Login the system	F3.4	T1
2	View reservation status	F3.5	T2
3	View train schedule	F3.6	T3
4	Reserve seat	F3.7	T4
5	Cancel seat	F3.8	T5
6	Exit the system	F3.9	T6

- **Variances**
  Nil
- **Comprehensive Statement**
  All the test cases were tested except F3.7, F3.8, F3.9 according to the test plan.
- **Summary of Results**

S. No.	Functionality	Function ID in SRS	Test cases	Status
1	Login the system	F3.4	T1	Pass
2	View reservation status	F3.5	T2	Bug Found. Could not be resolved.
3	View train schedule	F3.6	T3	Pass

- **Evaluation**
  The functions F3.4, F3.6 have been tested successfully. Function F3.5 couldn't be tested as the bug has been found. The bug is that the PNR number entry has also accepted alphabetical entries as wrong

Figure 9.8 Test summary report

A test summary report contains the following components [56]:

- **Test summary report identifier**
- **Description** Identify the test items being reported in this report with the test case/procedure ID.
- **Variances** Mention any deviation from the test plans, test procedures, if any.

- **Comprehensive statement** The originator of this report compares the comprehensiveness of the testing efforts made with the test plans. It describes what has been tested according to the plan and what has not been covered.
- **Summary of results** All the results are mentioned here with the resolved incidents and their solutions. Unresolved incidents are also reported.
- **Evaluation** Mention the status of the test results. If the status is fail, then mention its impact and severity level.
- **Summary of activities** All testing execution activities and events are mentioned with resource consumption, actual task durations, etc.
- **Approvals** List the names of the persons who approve this document with their signatures and dates.

SUMMARY

Testing is not an intuitive process. It is a systematic well-defined process. Therefore, it needs complete management. There should be a hierarchy of testing persons in the test organization with well-defined roles. Testing activities start with proper planning and continue with detailed test design specifications to result specifications. The idea is to plan and document the steps of STLC according to which the tester works. The tester plans, design, executes the test cases, and reports the test results.

This chapter discusses the test organization with the hierarchy of every testing person. A general test plan's components have been described. A master plan including verification and validation plan is also needed for testing a software. The verification plan and validation test plan at every stage – unit test plan, integration test plan, and system test plan, have also been discussed in this chapter.

The test case specifications along with test design specifications are discussed for designing the test cases. After the test case execution, the test results should also be reported. Test reporting exists in three main documents namely – test log, test incident report, and test summary report. All these testware have been explained in detail with their specifications.

Let us review the important concepts described in this chapter:

- Project manager is a key person in the testing group who interacts with project management, quality assurance, and marketing staff.
- Test leader leads a team of test engineers who work at the leaf-level of the hierarchy.
- Test engineers work under the lead of the test leader and are responsible for designing, developing, and maintaining test cases.
- Junior test engineers are newly hired testers. They usually go for training to learn the test strategy, test process, and testing tools.
- A test plan is defined as a document that describes the scope, approach, resources, and schedule of intended testing activities.

- Master test plan provides the highest level description of verification and validation efforts and drives the testing at various levels.
- Unit test is provided by the module developer. He prepares a test harness to identify the interfaces between the unit to be tested and other units.
- Integration test plan specifies the necessary steps needed to integrate individual modules and test the integration. It helps the technical team to think through the logical sequence of integration activities.
- Function test plan specifies the requirements for a bare-minimum functioning of the system. The plan must be ready with a traceability matrix that maps every function to its requirement and a list of functions to be tested.
- A system test plan is a systematic approach for testing a system containing a detailed understanding of what the eventual workflow will be. For this system, test cases are divided into some categories (recovery, security, performance, compatibility, etc.), according to which the system test plan is described.
- Acceptance test plan must have all the acceptance criteria defined in one document. If they are not available, then prepare them and plan the acceptance test accordingly. Another point in acceptance testing plan is to decide the criticality of acceptance feature defined. It is necessary to define the criticality so as to ensure that the system does not pass the accepted test, if it has failed in high critical acceptance requirement.
- Test log is a record of the testing events that take place during a test.
- Test incident report keeps a log of any unexpected event that has occurred during the test, which needs further investigation to resolve the bug.
- Test summary report is about the tests executed for a specific test design specification. It can measure how much testing efforts have been applied for the test.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. The key elements of test management are _____.
   (a) test organization
   (b) test harness
   (c) test planning
   (d) test monitoring and assessment
2. _____ is at the top-level in the test group hierarchy.
   (a) test engineer
   (b) test manager
   (c) test leader
   (d) junior engineer
3. Verification plan includes _____.
   (a) unit test plan

(b) integration test plan
(c) function design plan
(d) none of the above

4. Validation plan includes _____.
   (a) unit test plan
   (b) integration test plan
   (c) function design plan
   (d) none of the above
5. Master schedule in V&V planning _____.
   (a) summarizes the V&V tasks and their relationship with the overall project
   (b) summarizes the resources needed to perform V&V tasks
   (c) identifies the organization responsible for performing V&V tasks
   (d) none of the above
6. Test log is _____.
   (a) an evaluation report prepared when the testing is over
   (b) a form of bug report
   (c) a record of the testing events that take place during test
   (d) none of the above
7. Test incident report is _____.
   (a) an evaluation report prepared when the testing is over
   (b) a form of bug report
   (c) a record of the testing events that take place during test
   (d) none of the above
8. Test summary report is _____.
   (a) an evaluation report prepared when the testing is over
   (b) a form of bug report
   (c) a record of the testing events that take place during test
   (d) none of the above
9. Test incident report is _____.
   (a) test procedure specification document
   (b) test result specification document
   (c) test design specification document
   (d) none of the above
10. _____ integration testing is preferred while planning integration test.
   (a) top-down
   (b) bottom-up
   (c) sandwich
   (d) none of the above

11. Test engineer is responsible for _____.
    (a) test planning the given tasks by the test manager
    (b) providing tool training, if needed
    (c) interacting with customer
    (d) designing the test cases
12. Junior test engineer _____.
    (a) participates in test design and execution with experienced test engineers
    (b) assigns testing tasks to test engineers who are working under him
    (c) supervises test engineers
    (d) none of the above
13. Test leader is _____.
    (a) responsible for making test strategies with detailed master planning and schedule
    (b) interacts with customer regarding quality issues
    (c) acquires all the testing resources including tools
    (d) assigns testing tasks to test engineers who are working under him
14. Test manager _____.
    (a) interacts with customer regarding quality issues
    (b) acquires all the testing resources including tools
    (c) designs test cases
    (d) develops test harness

Review Questions

1. Discuss the key components of test management.
2. What are the major activities of a testing group?
3. What is meant by testing group hierarchy? Explain the role of each member in this hierarchy.
4. Suppose you are working in the testing group of a company. Identify your role in the testing group hierarchy. What are the major duties performed by you there?
5. What type of test planning will you plan for a real-time critical software? Design a test plan using test plan components.
6. How can the users/clients help in preparing the test plans?
7. What are the major activities in V&V planning?
8. Acquire SRS and SDD of any project and develop the following:
    (a) Unit test plan
    (b) Integration test plan
    (c) System test plan

9. Explain the importance of test harness in the integration test plan.
10. What is the difference between system test plan and acceptance test plan?
11. Discuss the role of traceability matrix in designing the test cases.
12. Discuss the role of test log, test incident report, and test summary report in validation testing.
13. Take three modules of your choice in a project and prepare the following for each module:
    (a) Test design specifications
    (b) Test case specifications
    (c) Test procedure specifications
    (d) Test result specifications
14. The modules taken in Question 13 need to be integrated. Which test planning will you choose for integrating the modules?
15. What is functional decomposition?
16. List all the testing deliverables (documents) and describe the purpose of each, in the organization where you are working.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter 10

Software Metrics

OBJECTIVES

After reading this chapter, you should be able to understand:

- Software projects need measurement to quantify things for better monitoring and control over software development
- Software metrics are quantitative measurements
- Various types of software metrics
- Entities to be measured for the software: process, product, and resource
- Recognition of attributes before measurement, as they are important for designing software metrics
- Line-of-code metrics
- Halstead metrics
- Function point analysis metrics

Today, every technical process demands measurement. We cannot describe a product by simply saying that it should be of high quality or that it should be reliable. Today, the demand is to quantify the terms 'good or high quality', 'more reliable', 'efficient', etc. It is a world of quantification. This becomes more important in terms of software, as its development is becoming more and more complex.

With the increase in complexity, it becomes important to control the software development process to monitor its progress on factors like time, budget, and resource constraints, and measure the quality of the end-product. Thus, if we want to control all these parameters in software, we need software metrics.

DeMarco rightly said, *You cannot control what you cannot measure*. Sometimes, we don't know what can derail a project, therefore it is essential to measure and record the characteristics of good projects as well as bad ones. Therefore, quantification has become a necessity for the effective management of the software process.

Software metrics are used by the software industry to quantify the development, operation, and maintenance of software. Metrics give us information regarding the status of an attribute of the software and help us to evaluate it in an objective way. The practice of applying software metrics to a software process and to a software product is a complex task. By evaluating an attribute of the software, we can know its status. From there, we can identify and classify what its situation is; which helps us to find opportunities of improvements in the software. This also helps us to make plans for modifications that need to be implemented in the future.

Software metrics play an important role in measuring the attributes that are critical for the success of a software project. Measurement of these attributes helps to make the characteristics and relationships between the attributes clearer. This in turn supports informed decision-making. Measuring the attributes of a development process enables the management to have a better insight into the process. Thus, measurement is a tool through which the management identifies important events and trends, enabling them to make informed decisions. Moreover, measurements help in predicting outcomes and evaluating risks, which in turn decreases the probability of unanticipated surprises in different processes.

10.1 Need of Software Measurement

Measurements are a key element for controlling software engineering processes. By controlling, it is meant that one can assess the status of the process, observe the trends to predict what is likely to happen, and take corrective action for modifying our practices. Measurements also play their part in increasing our understanding of the process by making visible relationships among process activities and entities involved. Lastly, measurements improve our processes by modifying the activities based on different measures.

Figure 10.1 Need for software metrics

On the basis of this discussion, software measurement is needed for the following activities (see Fig. 10.1):

Understanding Metrics help in making the aspects of a process more visible, thereby giving a better understanding of the relationships among the activities and entities they affect.

Control Using baselines, goals, and an understanding of the relationships, we can predict what is likely to happen and correspondingly, make appropriate changes in the process to help meet the goals.

Improvement By taking corrective actions and making appropriate changes, we can improve a product. Similarly, based on the analysis of a project, a process can also be improved.

10.2 Definition of Software Metrics

Software metrics can be defined as 'the continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products.'

The IEEE Standard Glossary of Software Engineering Terms [21] defines a metric as 'a quantitative measure of the degree to which a system component or process possesses a given attribute.'

10.3 Classification of Software Metrics

10.3.1 Product vs. Process Metrics

Software metrics may be broadly classified as either *product metrics* or *process metrics*. Product metrics are measures of the software product at any stage of its development, from requirements to installed system. Product metrics may measure the complexity of the software design, the size of the final program, or the number of pages of documentation produced.

Process metrics, on the other hand, are measures of the software development process, such as the overall development time, type of methodology used, or the average level of experience of the programming staff.

10.3.2 Objective vs. Subjective Metrics

Objective measures should always result in identical values for a given metric, as measured by two or more qualified observers. For subjective measures, even qualified observers may measure different values for a given metric. For example, for product metrics, the size of product measured in line of code (LOC) is an objective measure. In process metrics, the development time is an example of objective measure, while the level of a programmer's experience is likely to be a subjective measure.

10.3.3 Primitive vs. Computed Metrics

Primitive metrics are those metrics that can be directly observed, such as the program size in LOC, the number of defects observed in unit testing, or the

total development time for the project. Computed metrics are those that cannot be directly observed but are computed in some way from other metrics. For example, productivity metrics like LOC produced per person-month or product quality like defects per thousand lines of code.

10.3.4 Private vs. Public Metrics

This classification is based on the use of different types to process data. Because it is natural that individual software engineers might be sensitive to the use of metrics collected on an individual basis, these data should be private to individuals and serve as an indicator for individuals only. Examples of private metrics include defect rates (by individual and by module) and errors found during development.

Public metrics assimilate information that originally was private to individuals and teams. Project-level defect rates, effort, calendar times, and related data are collected and evaluated in an attempt to uncover indicators that can improve organizational process performance.

10.4 Entities to be Measured

In order to measure, it is needed to identify an entity and a specific attribute of it. It is very important to define clearly what is being measured, otherwise, the measures cannot be performed or the measures obtained can have different meaning for different people.

The entities considered in software measurement are:

- ***Processes*** Any activity related to software development.
- ***Product*** Any artifact produced during software development.
- ***Resource*** People, hardware, or software needed for a process.

The attributes of an entity can be internal or external.

- ***Internal attributes*** of any entity can be measured only based on the entity and therefore, measured directly. For example, size is an internal attribute of any software measurement.
- ***External attributes*** of any entity can be measured only with respect to how the entity is related with the environment and therefore, can only be measured indirectly. For example, reliability, an external attribute of a program, does not depend only on the program itself but also on the compiler, machine, and user. Productivity, an external attribute of a person, clearly depends on many factors such as the kind of process and the quality of the software delivered.

10.5 Size Metrics

The software size is an important metric to be used for various purposes. At the same time, it is difficult to measure because, unlike other physical products, a software cannot be measured directly with conventional units. Various approaches used for its measurement are given below.

10.5.1 Line of Code (LOC)

This metric is based on the number of lines of code present in the program. The lines of code are counted to measure the size of a program. The comments and blank lines are ignored during this measurement. The LOC metric is often presented on thousands of lines of code (KLOC). It is often used during the testing and maintenance phases, not only to specify the size of the software product, but also it is used in conjunction with other metrics to analyse other aspects of its quality and cost.

10.5.2 Token Count (Halstead Product Metrics)

The problem with LOC is that it is not consistent, because all lines of code are not at the same level. Some lines are more difficult to code than others. Another metric set has been given by Halstead. He stated that any software program could be measured by counting the number of operators and operands. From these set of operators and operands, he defined a number of formulae to calculate the vocabulary, the length, and the volume of the software program. Halstead extended this analysis to determine the effort and time. Some Halstead metrics are given below.

Program Vocabulary

It is the number of unique operators plus the number of unique operands as given below:

$$n = n1 + n2$$

where n = program vocabulary

$n1$ = number of unique operators

$n2$ = number of unique operands

Program Length

It is the total usage of all the operators and operands appearing in the implementation. It is given as,

$$N = N1 + N2$$

where N = program length

$N1$ = all operators appearing in the implementation
$N2$ = all operands appearing in the implementation

Program Volume

The volume refers to the size of the program and it is defined as the program length times the logarithmic base 2 of the program vocabulary. It is given as,

$$V = N \log_2 n$$

where V = program volume
N = program length
n = program vocabulary

10.5.3 Function Point Analysis (FPA)

It is based on the idea that the software size should be measured according to the functionalities specified by the user. Therefore, FPA is a standardized methodology for measuring various functions of a software from the user's point of view.

The size of an application is measured in function points. The process of counting the functions using FPA has been standardized by the International Function Point Users Group (IFPUG). IFPUG [123] has defined the rules and standards for calculating function points and it also promotes their use and evolution.

Process to Calculate Function Points

The process used to calculate the function points is given below [122]:

1. Determine the type of project for which the function point count is to be calculated. For example, development project (a new project) or enhancement project.
2. Identify the counting scope and the application boundary.
3. Identify data functions (internal logical functions and external interface files) and their complexity.
4. Identify transactional functions (external inputs, external outputs, and external queries) and their complexity.
5. Determine the unadjusted function point count (UFP).
6. Determine the value adjustment factor, which is based on 14 general system characteristics (GSCs).
7. Calculate the adjusted function point count (AFP).

Function point counting has been depicted in Fig. 10.2. This figure shows all the data functions and transactional functions. The details of all these functions are described in the following sections.

Figure 10.2 Functionality recognized in function point counting

Sizing Data Functions

Data functions are those functions in the project which relate to the logical data stored and available for update, reference, and retrieval. These functions are categorized in two parts: internal logical functions and external interface files. The categories are logical groupings of logically related data, and not physical representations. These are discussed below.

- ***Internal logical files (ILF)*** An internal logical file is a user-identifiable group of logically related data or control information maintained within the boundary of the application.
- ***External interface files (EIF)*** An external interface file is a user-identifiable group of logically related data or control information referenced by the application but maintained within the boundary of a different application. An EIF counted for an application must be in another application.

The physical count of ILFs and EIFs, together with the relative functional complexity of each, determines the contribution of the data function types to the unadjusted function point. Each identified data function must be assigned a functional complexity based on the number of data element types (DETs) and record element types (RETs) associated with the ILF and EIF.

DETs are unique user-recognizable, non-repeatable fields or attributes. RETs are user-recognizable subgroups (optional or mandatory) of data elements contained within an ILF or EIF. Subgroups are typically represented in an entity relationship diagram as entity subtypes or attribute entities.

Functional complexity is the rating assigned to each data function. The rating is based on record types and DETs being counted, as shown in Table 10.1.

Table 10.1 Complexity matrix for ILFs and EIFs

RETs	DETs		
	1–19	20–50	>=51
1	Low	Low	Average
2–5	Low	Average	High
>5	Average	High	High

Sizing Transactional Functions

Information systems are usually developed to mechanize and automate manual tasks. The tasks that have been automated are identified as transactional functions which represent the functionality provided to the user for processing the data by an application. The categorization of these functions is given below.

External inputs (EIs) are incoming data or control information to alter the behaviour of a system, e.g. adding, changing, deleting, etc.

External inquiries (EQs) send data outside the application through retrievals of data or control information from ILFs and/or EIFs, e.g. retrieval and display of a current description from our file.

External outputs (EOs) send data outside the application with processing logic other than or in addition to retrieval of data or control information, e.g. report of monthly sales with calculation by category.

Complexity and Contribution

The physical count of EIs, EOs, and EQs together with the relative functional complexity for each, determines the contribution of external inputs, outputs, and queries to the unadjusted function point count. Each identified EI, EO, and EQ must be assigned a functional complexity based on the number of DETs and FTRs (file type referenced) associated with them. Complexity matrices for EIs, EOs, and EQs are shown in Tables 10.2, 10.3, and 10.4.

Table 10.2 Complexity matrix for EIs

FTRs	DETs		
	1–4	5–15	>=16
<2	Low	Low	Average
2	Low	Average	High
>2	Average	High	High

Table 10.3 Complexity matrix for EOs

FTRs	DETs		
	1–5	6–19	>=20
<2	Low	Low	Average
2–3	Low	Average	High
>3	Average	High	High

Table 10.4 Complexity matrix for EQs

FTRs	DETs		
	1–5	6–19	>=20
<2	Low	Low	Average
2–3	Low	Average	High
>3	Average	High	High

Calculating Unadjusted Function Point (UFP)

The steps for calculating UFP are given below:

- Count all DETs/FTRs for all five components, i.e. ILF, EIF, EI, EO, and EQ of an application and determine the level of the component as low, average, or high, based on the individual complexity matrix, as described above.
- Count the number of all five components. The sum of each count is multiplied by an appropriate weight using Table 10.5, as shown below.

Table 10.5 IFPUGs unadjusted function point table

Components	Function Levels		
	Low	Average	High
ILF	X7	X10	X15
EIF	X5	X7	X10
EI	X3	X4	X6
EO	X4	X5	X7
EQ	X3	X4	X6

- Add all the five results calculated in the previous step. This is the final UFP.

Calculating Adjusted Function Point

There are some characteristics regarding every project which must be considered for the effort being measured for the project. Therefore, the function point calculated above is unadjusted, as the project dependent characteristics have not been considered in the function count. Therefore, IFPUG defines a

value adjustment factor (VAF) which is used as a multiplier of the unadjusted function point count in order to calculate the adjusted function point (AFP) count of an application.

Each GSC is evaluated in terms of its degree of influence (DI) on a scale of 0 to 5, as given below in Table 10.6.

Table 10.6 Degree of influence scaling

DI	Meaning
0	Not present, or no influence
1	Incidental influence
2	Moderate influence
3	Average influence
4	Significant influence throughout

Table 10.7 shows the GSCs to calculate the VAF.

Table 10.7 Components of VAF

Factor	Meaning
F1	Data communications
F2	Performance
F3	Transaction rate
F4	End user efficiency
F5	Complex processing
F6	Installation ease
F7	Multiple sites
F8	Distributed data processing
F9	Heavily used configuration
F10	Online data entry
F11	Online update
F12	Reusability
F13	Operational ease
F14	Facilitate change

The 14 GSCs shown in the table are summarized in the VAF. After applying the VAF, the UFP adjusts by 35% to determine the adjusted function point. The following steps determine the AFP.

- Evaluate the 14 GSCs on a scale of 0–5 to determine the DI for each GSC description.
- Add the DIs for all 14 GSCs to produce the total degree of influence (TDI).

- Use the TDI in the following equation to compute VAF.

$$VAF = (TDI \times 0.01) + 0.065$$

- The final adjusted function point is calculated as,

$$AFP = UFP \times VAF$$

Example 10.1

Consider a project with the following parameters: EI = 50, EO = 40, EQ = 35, ILF = 06, and ELF = 04. Assume all weighing factors are average. In addition, the system requires critical performance, average end-user efficiency, moderate distributed data processing, and critical data communication. Other GSCs are incidental. Compute the function points using FPA.

Solution

$$UFP = 50 \times 4 + 40 \times 5 + 35 \times 4 + 6 \times 10 + 4 \times 7 = 628$$

$$TDI = (4 + 3 + 2 + 4) \times 10 = 130$$

$$VAF = 130 \times 0.01 + 0.065 = 1.365$$

$$AFP = 628 \times 1.365 = 857.22$$

SUMMARY

Software measurement is not only useful but necessary. After all, how do you decide that a project is under control, if there is no measure for this? Software metrics play an important role today in understanding, controlling, and improving the software process as well as the product.

This chapter discusses the need for software measurement with its various types of metrics. Function point analysis is the major size metric used to measure the size of software. The technique has been adopted for measuring the development and testing effort. The chapter discusses the whole process of calculating the function points for a project to be developed. This chapter makes ground for the next chapter which is about software testing metrics.

Let us review the important concepts described in this chapter:

- Software metrics play an important role in measuring attributes that are critical to the success of a software project. Measurement of these attributes helps to make the characteristics and relationships between the attributes clearer.
- Software metric can be defined as a quantitative measure of the degree to which a system component or process possesses a given attribute.
- LOC (line of code) metric is based on the number of lines of code present in the program. The lines of code are counted for measuring the size of the program.
- Token count metric is based on counting the number of operators and operands in a program. From this set of operators and operands, a number of formulae are defined to calculate the vocabulary, the length, and the volume of the software program.

- Function point analysis (FPA) is a standardized methodology for measuring the various functions of a software from the user's point of view. The size of an application is measured in function points.
- Data functions are those functions in the project which relate to logical data stored and available for update, reference, and retrieval.
- An internal logical file (ILF) is a user-identifiable group of logically related data or control information, maintained within the boundary of the application.
- An external logical file (EIF) is a user-identifiable group of logically related data or control information, referenced by the application, but maintained within the boundary of a different application. An EIF counted for an application must be in another application.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Metrics give us information regarding the status of _____ in a software.
   (a) attributes
   (b) processes
   (c) costs
   (d) none of the above
2. _____ activity does not require software measurement.
   (a) Development
   (b) Understanding
   (c) Control
   (d) Improvement
3. Primitive metrics are those metrics that can be _____observed.
   (a) directly
   (b) indirectly
   (c) user
   (d) none of the above
4. Data functions are those functions in the project which relate to _____ data stored.
   (a) physical
   (b) logical
   (c) complex
   (d) none of the above
5. VAF = (TDI × 0.01) + _____
   (a) 0.076
   (b) 0.075
   (c) 0.065
   (d) none of the above

6. VAF adjusts the UFP by _____ to determine the adjusted function point.
   (a) 34%
   (b) 35%
   (c) 43%
   (d) none of the above
7. AFP = UFP __ VAF
   (a) +
   (b) *
   (c) –
   (d) /

Review Questions

1. What is the need for software measurement?
2. Discuss the various types of software metrics.
3. What is the disadvantage of LOC metrics?
4. What is the basis of Halstead metrics to calculate the size of a software?
5. What type of projects can be best counted with function point analysis?
6. Explain the process of calculating function points for a project which you are going to build.
7. Can you adopt the FPA for calculating the function points of a real-time software?
8. What is the difference between UFP and AFP?
9. Consider a project with the following parameters: EI= 60, EO= 40, EQ = 45, ILF = 06, ELF = 08. Assume all weighing factors are average. In addition, the system requires significant data communications, performance is very critical, designed code may be moderately reusable, and other GSCs are average. Compute the function points using FPA.
10. Consider a project with the following components: EI (simple) = 30, EO (average) = 20, EQ (average) = 35, ILF (complex) = 08, ELF (complex) = 05. In addition, the system requires significant end-user efficiency, moderate distributed data processing, critical data communications, and other GSCs are incidental. Compute the function points for this system using FPA.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

11

Testing Metrics for Monitoring and Controlling the Testing Process

OBJECTIVES

After reading this chapter, you should be able to understand:

- Testing metrics are important to monitor and control all testing activities
- Attributes for software testing on the basis of which testing metrics are designed
- Progress metrics
- Quality metrics
- Cost metrics
- Size metrics
- Testing effort estimation models: Halstead model, development ratio method, project-staff ratio method, test procedure method, task planning method, function point method, test-point method

Software testing is the most time-consuming part of SDLC. Therefore, managing the software testing process is a real challenge. Everyone in the project team has some questions regarding the testing process.

- How efficient is our testing?
- Is it worth doing verification as opposed to validation?
- What is the thoroughness of validation testing?
- What kind of errors are we going to detect and how many?
- How many errors remain in the product, even after testing?
- When should we stop testing?

Measurement answers all these questions. Test execution can be tracked effectively using software metrics. Software metrics play a vital role in understanding, controlling, and improving the test process. The organizations that can control their software testing processes are able to predict costs and schedules and increase the effectiveness, efficiency, and profitability of their business [113]. Therefore, knowing and measuring what is being done is important for an effective testing effort [112].

Thus, assessment of the effectiveness of a software testing process has to rely on appropriate measures. If these measures are embedded in the organizational-level testing strategy, they make the underlying testing process activities visible. It helps the managers and engineers to better acknowledge the connections among various process activities. Moreover, measurement helps in improving the software testing procedures, methods, tools, and activities, by providing objective evidence for evaluations and comparisons.

The benefits of measurement in software testing are [35]:

- Identifying the strengths and weaknesses of testing
- Providing insights into the current state of the testing process
- Evaluating testing risks
- Benchmarking
- Improving the planning process
- Improving the effectiveness of testing
- Evaluating and improving the product quality
- Measuring the productivity
- Determining the level of customer involvement and satisfaction
- Supporting, controlling, and monitoring the testing process
- Comparing the processes and products with those inside and outside the organization

11.1 Measurement Objectives for Testing

The objectives for assessing a test process should be well-defined. A number of metrics programs begin by measuring what is convenient or easy to measure, rather than by measuring what is needed. Such programs often fail because the resulting data are not useful to anyone [32]. A measurement program will be more successful, if it is designed with the goals of the project in mind. For this purpose, we can take the help of a *goal question metric* (GQM), first suggested by Basili and Rombach.

The GQM approach is based on the fact that the objectives of measurement should be clear, much before the data collection begins. According to this approach, the objectives of a measurement process should be identified first. The GQM approach, with reference to test process measurement, provides the following framework:

- Lists the major goals of the test process.
- Derives from each goal, the questions that must be answered to determine if the goals are being met.
- Decides what must be measured in order to answer the questions adequately.

In this way, we generate only those measures that are related to the goals. In many cases, several measurements may be required to answer a single question. Likewise, a single measurement may apply to more than one question.

Software measurement is effective only when the metrics are used and analysed in conjunction with one another. We will now discuss the metrics for testing.

11.2 Attributes and Corresponding Metrics in Software Testing

An organization needs to have a reference set of measurable attributes and corresponding metrics that can be applied at various stages during the course of execution of an organization-wide testing strategy. The attributes to be measured depend on the following factors:

- Time and phase in the software testing lifecycle.
- New business needs.
- Ultimate goal of the project.

Therefore, we discuss measurable attributes for software testing and the corresponding metrics based on the attribute categorization done by Wasif Afzal and Richard Torkar [121], as shown in Table 11.1. These attributes are discussed in the subsequent sections.

Table 11.1 Attribute categories

Category	Attributes to be Measured
Progress	• Scope of testing • Test progress • Defect backlog • Staff productivity • Suspension criteria • Exit criteria
Cost	• Testing cost estimation • Duration of testing • Resource requirements • Training needs of testing group and tool requirement • Cost-effectiveness of automated tool
Quality	• Effectiveness of test cases • Effectiveness of smoke tests • Quality of test plan • Test completeness
Size	• Estimation of test cases • Number of regression tests • Tests to automate

11.3 Attributes

11.3.1 Progress

The following attributes are defined under the progress category of metrics:

- Scope of testing
- Test progress
- Defect backlog
- Staff productivity
- Suspension criteria
- Exit criteria

Scope of testing It helps in estimating the overall amount of work involved, by documenting which parts of the software are to be tested [114], thereby estimating the overall testing effort required. It also helps in identifying the testing types that are to be used for covering the features under test.

Tracking test progress The progress of testing should also be measured to keep in line with the schedule, budget, and resources. If we monitor the progress with these metrics by comparing the actual work done with the planning, then the corrective measures can be taken in advance, thereby controlling the project.

For measuring the test progress, well-planned test plans are prepared wherein testing milestones are planned. Each milestone is scheduled for completion during a certain time period in the test plan. These milestones can be used by the test manager to monitor how the testing efforts are progressing. For example, executing all planned unit tests is one example of a testing milestone. Measurements need to be available for comparing the planned and actual progress towards achieving the testing milestones. To execute the testing milestones of all planned unit tests, the data related to the number of planned unit tests currently available and the number of executed unit tests on this date should be available.

The testing activity being monitored can be projected using graphs that show the trends over a selected period of time. The test progress *S* curve compares the test progress with the plan to indicate corrective actions, in case the testing activity is falling behind schedule. The advantage of this curve is that schedule slippage is difficult to ignore. The test progress *S* curve shows the following set of information on one graph [42]:

- Planned number of test cases to be completed successfully by a week.
- Number of test cases attempted in a week.
- Number of test cases completed successfully by a week.

Defect backlog It is the number of defects that are outstanding and unresolved at one point of time with respect to the number of defects occurring. The defect backlog metric is to be measured for each release of a software product for release-to-release comparisons [42]. One way of tracking it is to use *defect removal percentage* [112] for a prior release so that decisions regarding additional testing can be wisely taken in the current release. If the backlog increases, the bugs should be prioritized according to their criticality.

Staff productivity Measurement of testing staff productivity helps to improve the contributions made by the testing professionals towards a quality testing process. For example, as test cases are developed, it is interesting to measure the productivity of the staff in developing these test cases. This estimation is useful for the managers to estimate the cost and duration for testing activities.

The basic and useful measures of a tester's productivity are [60]:

- Time spent in test planning.
- Time spent in test case design.
- Number of test cases developed.
- Number of test cases developed per unit time.

Suspension criteria These are the metrics that establish conditions to suspend testing. It describes the circumstances under which testing would stop temporarily. The suspension criteria of testing describes in advance that the occurrence of certain events/conditions will cause the testing to stop temporarily. With the help of suspension metrics, we can save precious testing time by raising the conditions that suspend testing. Moreover, it indicates the shortcomings of the application and the environmental setup.

The following conditions indicate the suspension of testing:

- There are incomplete tasks on the critical path.
- Large volumes of bugs are occurring in the project.
- Critical bugs.
- Incomplete test environments (including resource shortages).

Exit criteria The exit criteria are established for all the levels of a test plan. It indicates the conditions that move the testing activities forward from one level to the next. If we are clear when to exit from one level, the next level can start. On the other hand, if the integration testing exit criteria are not strict, then system testing may start before the completion of integration testing.

The exit criteria determine the termination of testing effort which must be communicated to the development team prior to the approval of the test plan.

The exit criteria should also be standardized. If there are any open faults, then the software development manager, along with the project manager and the members of the change control board, decides whether to fix the faults, or defer them to the next release, or take the risk of shipping it to the customer with the faults.

The following metrics are studied for the exit criteria of testing:

- Rate of fault discovery in regression tests.
- Frequency of failing fault fixes.
- Fault detection rate.

11.3.2 Cost

The following attributes are defined under the cost category of metrics:

- Testing cost estimation
- Duration of testing
- Resource requirements
- Training needs of testing groups and tool requirements
- Cost effectiveness of automated tools

Testing cost estimation The metrics supporting the budget estimation of testing need to be established early. It also includes the cost of test planning itself.

Duration of testing There is also a need to estimate the testing schedule during test planning. As a part of the testing schedule, the time required to develop a test plan is also estimated. A testing schedule contains the timelines of all testing milestones. The testing milestones are the major testing activities carried out according to a particular level of testing. A work break-down structure is used to divide the testing efforts into tasks and activities. The timelines of activities in a testing schedule matches the time allocated for testing in the overall project plan.

Resource requirements Test planning activities have to estimate the number of testers required for all testing activities with their assigned duties planned in the test plan.

Training needs of testing groups and tool requirements Since the test planning also identifies the training needs of testing groups and their tool requirements, we need to have metrics for training and tool requirements.

Cost-effectiveness of automated tools When a tool is selected to be used for testing, it is beneficial to evaluate its cost-effectiveness. The cost-effectiveness

is measured by taking into account the cost of tool evaluation, tool training, tool acquisition, and tool update and maintenance.

11.3.3 Quality

The following attributes are defined under this category of metrics:

- Effectiveness of test cases
- Effectiveness of smoke tests
- Quality of test plan
- Test completeness

Effectiveness of test cases The test cases produced should be effective so that they uncover more and more bugs. The measurement of their effectiveness starts from the test case specifications. The specifications must be verified at the end of the test design phase for conformance with the requirements. While verifying the specifications, emphasis must be given to those factors that affect the effectiveness of test cases including the test cases with incomplete functional specifications, poor test designs, and wrong interpretation of test specifications by the testers [110].

Common methods available for verifying test case specifications include inspections and traceability of test case specifications to functional specifications. These methods seldom help to improve the fault-detecting ability of test case specifications.

There are several measures based on faults to check the effectiveness of test cases. Some of them are discussed here:

1. *Number of faults* found in testing.
2. *Number of failures* observed by the customer which can be used as a reflection of the effectiveness of test cases.
3. *Defect-removal efficiency* is another powerful metric for test-effectiveness, which is defined as the ratio of the number of faults actually found in testing and the number of faults that could have been found in testing. There are potential issues that must be taken into account while measuring the defect-removal efficiency. For example, the severity of bugs and an estimate of time by which the customers would have discovered most of the failures are to be established. This metric is more helpful in establishing the test effectiveness in the long run as compared to the current project.
4. *Defect age* is another metric that can be used to measure the test effectiveness, which assigns a numerical value to the fault, depending on the phase

in which it is discovered. Defect age is used in another metric called *defect spoilage* to measure the effectiveness of defect-removal activities. Defect spoilage is calculated as [117]:

$$\text{Spoilage} = \frac{\text{Sum of (Number of defects} \times \text{Defect age)}}{\text{Total number of defects}}$$

Spoilage is more suitable for measuring the long-term trend of test-effectiveness. Generally, low values of defect spoilage mean more effective defect discovery processes. The effectiveness of a test case can also be judged on the basis of the coverage provided. It is a powerful metric in the sense that it is not dependent on the quality of software and also, it is an in-process metric that is applicable when the testing is actually done.

Example 11.1

Consider a project with the following distribution of data and calculate its defect spoilage.

SDLC phase	No. of defects	Defect age
Requirement Specs.	34	2
HLD	25	4
LLD	17	5
Coding	10	6

Solution

$$\text{Spoilage} = (34 \times 2 + 25 \times 4 + 17 \times 5 + 10 \times 6)/86 = 3.64$$

Effectiveness of smoke tests Smoke tests are required to ensure that the application is stable enough for testing, thereby assuring that the system has all the functionality and is working under normal conditions. This testing does not identify faults, but establishes confidence over the stability of a system. The tests that are included in smoke testing cover the basic operations that are most frequently used, e.g. logging in, addition, and deletion of records. Smoke tests need to be a subset of the regression testing suite. It is time-consuming and human-intensive to run the smoke tests manually, each time a new build is received for testing. Automated smoke tests are executed against the software, each time the build is received, ensuring that the major functionality is working.

Quality of test plan The quality of the test plan produced is also a candidate attribute to be measured, as it may be useful in comparing different plans for different products, noting down the changes observed, and improving the test plans for the future. The test plan should also be effective in giving a high

number of errors. Thus, the quality of a test plan is measured in concern with the probable number of errors.

To evaluate a test plan, Berger describes a multi-dimensional qualitative method using *rubrics* [118]. Rubrics take the form of a table, where each row represents a particular dimension and each column represents a ranking of the dimension. There are ten dimensions that contribute to the philosophy of test planning. These ten dimensions are:

1. Theory of objective
2. Theory of scope
3. Theory of coverage
4. Theory of risk
5. Theory of data
6. Theory of originality
7. Theory of communication
8. Theory of usefulness
9. Theory of completeness
10. Theory of insightfulness

Against each of these theories, there is a rating of excellent, average, and poor, depending on certain criteria, as shown in Table 11.2.

Table 11.2 Measuring test plan through rubrics

Dimension	Rating		
	Excellent	**Average**	**Poor**
Theory of Objective	Identification of realistic test objectives to have the most efficient test cases.	Describes somewhat credible test objectives.	No objectives or irrelevant objectives.
Theory of Scope	Specific and unambiguous test scope.	Some scope of testing understood by only some people.	Wrong scope assumptions.
Theory of Coverage	The test coverage is completely relative to the test scope.	The test coverage is somewhat related to the test scope.	Coverage is not related to test scope or objective.
Theory of Risk	Identifies possible testing risks.	Risks have inappropriate priorities.	No understanding of project risk.
Theory of Data	Efficient method to generate enough valid and invalid test data.	Some method to generate test data.	There is no method of capturing data.

Theory of Originality	High information in test plan.	A test plan template has some original content.	Template has not been filled in.
Theory of Communication	Multiple modes to communicate test plan to, and receive feedback from appropriate stakeholders.	Less effective feedback.	No distribution, no opportunity for feedback
Theory of Usefulness	Effective test plan such that it discovers critical bugs early.	Critical bugs are not discovered earlier.	Test plans are not useful to the organization.
Theory of Completeness	Enough testing through the plan.	Most test items have been identified.	Enough testing criteria has not been defined.
Theory of Insightfulness	Understanding of what is interesting and challenging in testing this specific project.	The test plan is effective, but weak.	No insight through the plan.

Measuring test completeness This attribute refers to how much of code and requirements are covered by the test set. The advantages of measuring test coverage are that it provides the ability to design new test cases and improve existing ones. There are two issues: (i) whether the test cases cover all possible output states and (ii) the adequate number of test cases to achieve test coverage. The relationship between code coverage and the number of test cases is described by the following expression [119]:

$$C(x) = 1 - e^{-(p/N) * x}$$

where $C(x)$ is the coverage after executing x number of test cases, N is the number of blocks in the program, and p is the average number of blocks covered by a test case during the function test. The function indicates that test cases cover some blocks of code more than others, while increasing test coverage beyond a threshold is not cost-effective.

Test coverage for control flow and data flow can be measured using spanning sets of entities in a program flow graph. A spanning set is a minimum subset of the set of entities of the program flow graph such that a test suite covering the entities in this subset is guaranteed to cover every entity in the set [120].

At the system testing level, we should measure whether all the features of the product are being tested or not. Common requirements coverage metric is the percentage of requirements covered by at least one test [112]. A requirements traceability matrix can be used for this purpose.

11.3.4 Size

The following attributes are defined under the size category of metrics:

- Estimation of test cases
- Number of regression tests
- Tests to automate

Estimation of test cases To fully exercise a system and to estimate its resources, an initial estimate of the number of test cases is required. Therefore, metrics estimating the required number of test cases should be developed.

Number of regression tests Regression testing is performed on a modified program that establishes confidence that the changes and fixes against reported faults are correct and have not affected the unchanged portions of the program [51]. However, the number of test cases in regression testing becomes too large to test. Therefore, careful measures are required to select the test cases effectively.

Some of the measurements to monitor regression testing are [60]:

- Number of test cases re-used
- Number of test cases added to the tool repository or test database
- Number of test cases rerun when changes are made to the software
- Number of planned regression tests executed
- Number of planned regression tests executed and passed

Tests to automate Tasks that are repetitive in nature and tedious to perform manually are prime candidates for an automated tool. The categories of tests that come under repetitive tasks are:

- Regression tests
- Smoke tests
- Load tests
- Performance tests

If automation tools have a direct impact on the project, it must be measured. There must be benefits attached with automation including speed, efficiency, accuracy and precision, resource reduction, and repetitiveness. All these factors should be measured.

11.4 Estimation Models for Estimating Testing Efforts

After discussing the attributes needed to measure the testing efforts, we will see some estimation models and metrics that are being used.

11.4.1 Halstead Metrics

The metrics derived from Halstead measures described in Chapter 10 can be used to estimate testing efforts, as given by Pressman [116]. Halstead developed expressions for program volume V and program level PL which can be used to estimate testing efforts. The program volume describes the number of volume of information in bits required to specify a program. The program level is a measure of software complexity. Using these definitions, Halstead effort e can be computed as [124]:

$$PL = 1/[(n1/2) \times (N2/n2)]$$

$$e = V/PL$$

The percentage of overall testing effort to be allocated to a module k can be estimated using the following relationship:

$$\textbf{Percentage of testing effort } (k) = e(k)/\sum e(i)$$

where $e(k)$ is the effort required for module k and $\sum e(i)$ is the sum of Halstead effort across all modules of the system.

Example 11.2

In a module implementation of a project, there are 17 unique operators, 12 unique operands, 45 total operators, and 32 total operands appearing in the module implementation. Calculate its Halstead effort.

Solution

$$n = n1 + n2 = 17 + 12 = 92$$

$$N = N1 + N2 = 45 + 32 = 77$$

$$V = N \log_2 n = 77 \times \log_2 92 = 502.3175$$

$$\text{PL} = 1/[(n1/2) \times (N2/n2)] = 3/68 = 0.044$$

$$e = V/PL = 11416.30$$

11.4.2 Development Ratio Method

This model is based on the estimation of development efforts. The number of testing personnel required is estimated on the basis of the developer-tester ratio [12]. The results of applying this method is dependent on numerous factors including the type and complexity of the software being developed, testing level, scope of testing, test-effectiveness during testing, error tolerance level for testing, and available budget.

Another method of estimating tester-to-developer ratios, based on heuristics, is proposed by K. Iberle and S. Bartlett [115]. This method selects a baseline project(s), gathers testers-to-developers ratios, and collects data on various effects like developer-efficiency at removing defects before testing, developer-efficiency at inserting defects, defects found per person, and the value of defects found. After that, an initial estimate is made to calculate the number of testers based upon the ratio of the baseline project. The initial estimate is adjusted using professional experience to show how the above-mentioned effects affect the current project and the baseline project.

11.4.3 Project-Staff Ratio Method

Project-staff ratio method makes use of historical metrics by calculating the percentage of testing personnel from the overall allocated resources planned for the project [12]. The percentage of a test team size may differ according to the type of project. The template can be seen in Table 11.3.

Table 11.3 Project-staff ratio

Project type	Total number of project staff	Test team size %	Number of testers
Embedded system	100	23	23
Application development	100	8	8

11.4.4 Test Procedure Method

This model is based on the number of test procedures planned. The number of test procedures decides the number of testers required and the testing time [12]. Thus, the baseline of estimation here is the quantity of test procedures. But you have to do some preparation before actually estimating the resources. It includes developing a historical record of projects including the data related to size (e.g. number of function points, number of test procedures used) and test effort measured in terms of personnel hours. Based on the estimates of historical development size, the number of test procedures required for the new project is estimated.

The only thing to be careful about in this model is that the projects to be compared should be similar in terms of nature, technology, required expertise, and problems solved. The template for this model can be seen in Table 11.4. For a historical project, person-hours consumed for testing test procedures were observed and a factor in the form of number hours per test procedure was calculated. This factor is used for a new project where the number of test procedures are given. Using this factor, we will calculate the number of person-hours for the new project. Then, with the knowledge of the total expended period for this new project, we are able to calculate the number of testers required.

Table 11.4 Test procedure method

	Number of test procedures (NTP)	Number of person-hours consumed for testing (PH)	Number of hours per test procedure = PH/NTP	Total period in which testing is to be done (TP)	Number of testers = PH/TP
Historical Average Record	840	6000	7.14	10 months (1600 hrs)	3.7
New Project Estimate	1000	7140	7.14	1856 hrs	3.8

11.4.5 Task Planning Method

In this model, the baseline for estimation is the historical records of the number of personnel hours expended to perform testing tasks. The historical records collect data related to the work break-down structure and the time required for each task so that the records match the testing tasks [12]. First, the number of person-hours for the full project is calculated as in the test procedure method (see Table 11.5).

Table 11.5 Calculating number of person-hours

Number of Test Procedures (NTP)	Person-hours consumed for testing (PH)	Hours per test procedure = PH/NTP
840	6000	7.14
1000 (New project)	7140	7.14

The historical data is observed to see how much time has been consumed on individual test activities. After estimating the average idea on historical data, the total person-hours are distributed, which is later adjusted according to some conditions in the new project (see Table 11.6).

Table 11.6 Task planning method

Testing activity	Historical value	% time of the project consumed on the test activity	Preliminary estimate of person hours	Adjusted estimate of person-hours
Test planning	210	3.5	249	
Test design	150	2.5	178	
Test execution	180	3	214	
Project total	6000	100%	7140	6900

Next, the test team size is calculated using the total adjusted estimate of person-hours for the project, as given in Table 11.7.

Table 11.7 Test team size using task planning method

	NTP	PH (Adjusted estimate)	Number of hours per test procedure = PH/NTP	TP	Number of testers = PH/TP
New project estimate	1000	6900	6.9	1856 hrs	3.7

After getting an idea about the model for effort estimation, let us discuss some important metrics being used in various phases of testing.

11.5 Architectural Design Metric used for Testing

Card and Glass [57] introduced three types of software design complexity that can also be used in testing. These are discussed below.

Structural Complexity

It is defined as

$$S(m) = f^2_{out}(m)$$

where S is the structural complexity and $f_{out}(m)$ is the fan-out of module m.

This metric gives us the number of stubs required for unit testing of the module m. Thus, it can be used in unit testing.

Data Complexity

This metric measures the complexity in the internal interface for a module m and is defined as

$$D(m) = v(m) \,/\, [f_{\text{out}}(m) + 1]$$

where $v(m)$ is the number of input and output variables that are passed to and from module m.

This metric indicates the probability of errors in module m. As the data complexity increases, the probability of errors in module m also increases. For example, module X has 20 input parameters, 30 internal data items, and 20 output parameters. Similarly, module Y has 10 input parameters, 20 internal data items, and 5 output parameters. Then, the data complexity of module X is more as compared to Y, therefore X is more prone to errors. Therefore, testers should be careful while testing module X.

System Complexity

It is defined as the sum of structural and data complexity:

$$SC(m) = S(m) + D(m)$$

Since the testing effort of a module is directly proportional to its system complexity, it will be difficult to unit test a module with higher system complexity. Similarly, the overall architectural complexity of the system (which is the sum total of system complexities of all the modules) increases with the increase in each module's complexity. Consequently, the efforts required for integration testing increase with the architectural complexity of the system.

11.6 Information Flow Metrics used for Testing

Researchers have used information flow metrics between modules. For understanding the measurement, let us understand the way the data moves through a system:

- **Local direct flow** exists if
  - (i) a module invokes a second module and passes information to it.
  - (ii) the invoked module returns a result to the caller.
- **Local indirect flow** exists if the invoked module returns information that is subsequently passed to a second invoked module.
- **Global flow** exists if information flows from one module to another via a global data structure.

The two particular attributes of the information flow can be described as follows:

(i) **Fan-in** of a module *m* is the number of local flows that terminates at *m*, plus the number of data structures from which information is retrieved by *m*.

(ii) **Fan-out** of a module *m* is the number of local flows that emanate from *m*, plus the number of data structures that are updated by *m*.

11.6.1 Henry and Kafura Design Metric

Henry and Kafura's information flow metric is a well-known approach for measuring the total level of information flow between individual modules and the rest of the system. They measure the information flow complexity as

$$IFC(m) = length(m) \times ((fan\text{-}in(m) \times fan\text{-}out(m))^2$$

Higher the *IF* complexity of *m*, greater is the effort in integration and integration testing, thereby increasing the probability of errors in the module.

11.7 Cyclomatic Complexity measures for Testing

McCabe's cyclomatic complexity was discussed in the previous chapter. This measure can be used in software testing in the following ways:

- Since the cyclomatic number measures the number of linearly independent paths through flow graphs, it can be used as the set of minimum number of test cases. If the number of test cases is lesser than the cyclomatic number, then you have to search for the missing test cases. In this way, it becomes a thumb rule that cyclomatic number provides the minimum number of test cases for effective branch coverage.
- McCabe has suggested that ideally, cyclomatic number should be less than or equal to 10. This number provides a quantitative measure of testing difficulty. If the cyclomatic number is more than 10, then the testing effort increases due to the following reasons:
    1. Number of errors increase.
    2. Time required to detect and correct the errors increase.

Thus, the cyclomatic number is an important indication of the ultimate reliability. The amount of test design and test effort is better approximated by the cyclomatic number as compared to the lines of code (LOC) measure. A project in USA that applied these rules did achieve zero defects for at least 12 months after its delivery [34].

11.8 Function Point Metrics for Testing

The function point (FP) metric is used effectively for measuring the size of a software system. Function-based metrics can be used as a predictor for the overall testing effort. Various project-level characteristics (e.g. testing effort and time, errors uncovered, number of test cases produced) of past projects can be collected and correlated with the number of FP produced by a project team. The team can then project the expected values of these characteristics for the current project.

Listed below are a few FP measures:

1. Number of hours required for testing per FP.
2. Number of FPs tested per person-month.
3. Total cost of testing per FP.
4. Defect density measures the number of defects identified across one or more phases of the development project lifecycle and compares that value with the total size of the system. It can be used to compare the

density levels across different lifecycle phases or across different development efforts. It is calculated as

Number of defects (by phase or in total) / Total number of FPs

5. Test case coverage measures the number of test cases that are necessary to adequately support thorough testing of a development project. This measure does not indicate the effectiveness of test cases, nor does it guarantee that all conditions have been tested. However, it can be an effective comparative measure to forecast anticipated requirements for testing that may be required on a development system of a particular size. This measure is calculated as

Number of test cases / Total number of FPs

Capers Jones [146] estimates that the number of test cases in a system can be determined by the function points estimate for the corresponding effort. The formula is

$$\textit{Number of test cases} = (\textit{function points})^{1.2}$$

Function points can also be used to measure the acceptance test cases. The formula is

$$\textit{Number of test cases} = (\textit{function points}) \times 1.2$$

The above relationships show that test cases grow at a faster rate than function points. This is intuitive because, as an application grows, the number of inter-relationships within the applications becomes more complex. For example, if a development application has 1,000 function points, there should be approximately 4,000 total test cases and 1,200 acceptance test cases.

Example 11.3

In a project, the estimated function points are 760. Calculate the total number of test cases in the system and the number of test cases in acceptance testing. Also, calculate the defect density (number of total defects is 456) and test case coverage.

Solution

Total number of test cases $= (760)^{1.2} = 2864$

Number of test cases for acceptance testing $= (760) \times 1.2 = 912$

Defect density $= 456/760 = 0.6$

Test case coverage $= 2864/760 = 3.768$

11.9 Test Point Analysis (TPA)

Test point analysis is a technique to measure the black-box test effort estimation, proposed by Drs Eric P W M Van Veenendaal CISA and Ton Dekkers [111]. The estimation is for system and acceptance testing. The technique uses function point analysis in the estimation. TPA calculates the test effort estimation in test points for highly important functions, according to the user and also for the whole system. As the test points of the functions are measured, the tester can test the important functionalities first, thereby predicting the risk in testing. This technique also considers the quality characteristics, such as functionality, security, usability, efficiency, etc. with proper weightings.

11.9.1 Procedure for Calculating TPA

For TPA calculation, first the dynamic and static test points are calculated (see Fig. 11.1). *Dynamic test points* are the number of test points which are based on dynamic measurable quality characteristics of functions in the system. Dynamic test points are calculated for every function. To calculate a dynamic function point, we need the following:

- Function points (FPs) assigned to the function.
- Function dependent factors (FDC), such as complexity, interfacing, function-importance, etc.
- Quality characteristics (QC).

The dynamic test points for individual functions are added and the resulting number is the dynamic test point for the system.

Similarly, *static test points* are the number of test points which are based on static quality characteristics of the system. It is calculated based on the following:

- Function points (FPs) assigned to the system.
- Quality requirements or test strategy for static quality characteristics (QC).

After this, the dynamic test point is added to the static test point to get the total test points (TTP) for the system. This total test point is used for calculating *primary test hours* (PTH). PTH is the effort estimation for primary testing activities, such as preparation, specification, and execution. PTH is calculated based on the environmental factors and productivity factors. Environmental factors include development environment, testing environment, testing tools,

etc. Productivity factor is the measure of experience, knowledge, and skills of the testing team.

Secondary testing activities include management activities like controlling the testing activities. *Total test hours* (TTH) is calculated by adding some allowances to secondary activities and PTH. Thus, TTH is the final effort estimation for the testing activities.

Figure 11.1 Procedure for test point analysis

11.9.2 Calculating Dynamic Test Points

The number of dynamic test points for each function in the system is calculated as

$$DTP = FP \times FDC_w \times QC_{dw}$$

where DTP = number of dynamic test points

FP = function point assigned to function.

FDC_w = weight-assigned function-dependent factors

QC_{dw} = quality characteristic factor, wherein weights are assigned to dynamic quality characteristics

FDC_w is calculated as

$$FDC_w = ((FI_w + UIN_w + I + C) \div 20) \times U$$

where FI_w = function importance rated by users

UIN_w = weights given to usage intensity of the function, i.e. how frequently the function is being used

I = weights given to the function for interfacing with other functions, i.e. if there is a change in function, how many functions in the system will be affected

C = weights given to the complexity of function, i.e. how many conditions are in the algorithm of function

U = uniformity factor

The ratings done for providing weights to each factor of FDC_w can be seen in Table 11.8.

Table 11.8 Ratings for FDC_w factors

Factor/ Rating	Function Importance (FI)	Function usage Intensity (UIN)	Interfacing (I)	Complexity (C)	Uniformity Factor
Low	3	2	2	3	0.6 for the function, wherein test specifications are largely re-used such as in clone function or dummy function. Otherwise, it is 1.
Normal	6	4	4	6	
High	12	12	8	12	

QC_{dw} is calculated based on four dynamic quality characteristics, namely suitability, security, usability, and efficiency. First, the rating to every quality characteristic is given and then, a weight to every QC is provided. Based on the rating and weights, QC_{dw} is calculated.

$$QC_{dw} = \sum (rating\ of\ QC / 4) \times weight\ factor\ of\ QC$$

Rating and weights are provided based on the following characteristics, as shown in Table 11.9.

Table 11.9 Ratings and weights for QC_{dw}

Characteristic/ Rating	Not Important (0)	Relatively Unimportant (3)	Medium Importance (4)	Very Important (5)	Extremely Important (6)
Suitability	0.75	0.75	0.75	0.75	0.75
Security	0.05	0.05	0.05	0.05	0.05
Usability	0.10	0.10	0.10	0.10	0.10
Efficiency	0.10	0.10	0.10	0.10	0.10

11.9.3 Calculating Static Test Points

The number of static test points for the system is calculated as

$$STP = FP \times \sum QC_{sw} / 500$$

where STP = static test point
FP = total function point assigned to the system
QC_{sw} = quality characteristic factor, wherein weights are assigned to static quality characteristics

Static quality characteristic refers to what can be tested with a checklist. QC_{sw} is assigned the value 16 for each quality characteristic which can be tested statically using the checklist.

$$\textit{Total test points (TTP)} = DTP + STP$$

11.9.4 Calculating Primary Test Hours

The total test points calculated above can be used to derive the total testing time in hours for performing testing activities. This testing time is called primary test hours (PTH), as it is an estimate for primary testing activities like preparation, specification, execution, etc. This is calculated as

$$PTH = TTP \times \textit{productivity factor} \times \textit{environmental factor}$$

Productivity factor It is an indication of the number of test hours for one test point. It is measured with factors like experience, knowledge, and skill set of the testing team. Its value ranges between 0.7 to 2.0. But explicit weightage for testing team experience, knowledge, and skills have not been considered in this metric.

Environmental factor Primary test hours also depend on many environmental factors of the project. Depending on these factors, it is calculated as

$$\begin{aligned}\textit{Environmental factor} = {} & \textit{weights of (test tools + development testing} \\ & \textit{+ test basis + development environment} \\ & \textit{+ testing environment + testware)/21}\end{aligned}$$

These factors and their weights are discussed below.

Test tools It indicates the use of automated test tools in the system. Its rating is given in Table 11.10.

Table 11.10 Test tool ratings

1	Highly automated test tools are used.
2	Normal automated test tools are used.
4	No test tools are used.

Development testing It indicates the earlier efforts made on development testing before system testing or acceptance testing for which the estimate is being done. If development test has been done thoroughly, then there will be less effort and time needed for system and acceptance testing, otherwise, it will increase. Its rating is given in Table 11.11.

Table 11.11 Development testing ratings

2	Development test plan is available and test team is aware about the test cases and their results.
4	Development test plan is available.
8	No development test plan is available.

Test basis It indicates the quality of test documentation being used in the system. Its rating is given in Table 11.12.

Table 11.12 Test basis rating

3	Verification as well as validation documentation are available.
6	Validation documentation is available.
12	Documentation is not developed according to standards.

Development environment It indicates the development platforms, such as operating systems, languages, etc. for the system. Its rating is given in Table 11.13.

Table 11.13 Development environment rating

2	Development using recent platform.
4	Development using recent and old platform.
8	Development using old platform.

Test environment It indicates whether the test platform is a new one or has been used many times on the systems. Its rating is given in Table 11.14.

Table 11.14 Test environment rating

1	Test platform has been used many times.
2	Test platform is new but similar to others already in use.
4	Test platform is new.

Testware It indicates how much testware is available in the system. Its rating is given in Table 11.15.

Table 11.15 Testware rating

1	Testware is available along with detailed test cases.
2	Testware is available without test cases.
4	No testware is available.

11.9.5 Calculating Total Test Hours

Primary test hours is the estimation of primary testing activities. If we also include test planning and control activities, then we can calculate the total test hours.

Total test hours = PTH + Planning and control allowance

Planning and Control allowance is calculated based on the following factors:

Team size

3	≤ 4 team members
6	5–10 team members
12	>10 team members

Planning and control tools

2	Both planning and controlling tools are available.
4	Planning tools are available.
8	No management tools are available.

Planning and control allowance (%)
= weights of (team size + planning and control tools)

Planning and control allowance (hours)
= planning and control allowance (%) × PTH

Thus, the total test hours is the total time taken for the whole system. TTH can also be distributed for various test phases, as given in Table 11.16.

Table 11.16 TTH distribution

Testing phase	% of TTH
Plan	10%
Specification	40%
Execution	45%
Completion	5%

Example 11.4

Calculate the total test points for a module whose specifications are: functions points = 414, ratings for all FDC_w factors are normal, uniformity factor =1, rating for all QC_{dw} are 'very important' and for QC_{sw}, three static qualities are considered.

Solution

$$FDC_w = ((FI_w + UIN_w + I + C)/20) \times U = ((6+4+4+6)/20) \times 1 = 1$$

$$QC_{dw} = \sum (\text{rating of QC} / 4) \times \text{weight factor of QC}$$
$$= (5/4 \times 0.75) + (5/4 \times 0.05) + (5/4 \times 0.10) + (5/4 \times 0.10)$$
$$= 0.9375 + 0.0625 + 0.125 + 0.125 = 1.25$$

$$DTP = FP \times FDC_w \times QC_{dw}$$
$$= 414 \times 1 \times 1.25 = 517.5$$

$$STP = FP \times \sum QC_{sw} / 500$$
$$= 414 \times ((16 \times 3)/500) = 39.744$$

$$\text{Total test points (TTP)} = DTP + STP = 517.5 + 39.744 = 557.244$$

11.10 Some Testing Metrics

Everyone in the testing team wants to know when the testing should stop. To know when the testing is complete, we need to track the execution of testing. This is achieved by collecting data or metrics, showing the progress of testing. Using these progress metrics, the release date of the project can be determined. These metrics are collected iteratively during the stages of test execution cycle. Some metrics [12] are discussed below.

Test Procedure Execution Status

It is defined as:

Test Proc Exec. Status (%) = Number of executed test cases / Total number of test cases

This metric ascertains the number of percentage of test cases remaining to be executed.

Defect Aging

Defect aging is the turnaround time for a defect to be corrected. This metric is defined as

Defect aging = Closing date of bug – Start date when bug was opened

This metric data for various bugs is used for a trend analysis such that it can be known that *n* number of defects per day can be fixed by the test

team. For example, suppose 100 defects are being recorded for a project. If documented, the past experience indicates that the team can fix as many as 20 defects per day, the turnaround time for 100 problem reports may be estimated at one work-week.

Defect Fix Time to Retest

It is defined as

Defect fix time to retest
= Date of fixing the bug and releasing in new build – Date of retesting the bug

This metric provides a measure that the test team is retesting all the modifications corresponding to bugs at an adequate rate. If the rate of fixing and retesting them is not on time, then the project progress will suffer and consequently will increase the cost of the project.

Defect Trend Analysis

It is defined as the trend in the number of defects found as the testing life cycle progresses. For example,

Number of defects of each type detected in unit test per hour

Number of defects of each type detected in integration test per hour

To strengthen the defect information, defects of a type can also be classified according to their severity of impact on the system. It will be useful for testing, if we get many defects of high severity. For example,

Number of defects over severity level X (where X is an integer value for measuring the severity levels)

The trend can be such that the number of bugs increases or decreases as the system approaches its completion. Ideally, the trend should be such that the number of newly opened defects should decrease as the system testing phase ends. If the trend shows that the number of bugs are increasing with every stage of testing:

1. Previous bugs have not been closed/fixed properly.
2. Testing techniques for testing coverage is not adequate for earlier builds.
3. Inability to execute some tests until some of the defects have been fixed, allowing execution of those tests, which then find new defects.

Recurrence Ratio

This metric indicates the quality of bug-fixes. The quality of bug-fixes is good if it does not introduce any new bug in the previous working functionality of the software and the bug does not re-occur. But if the bug-fixing introduces new bugs in the already working software or the same bug re-occurs, then the quality is not good. Moreover, this metric indicates the test team the degree to

which the previously working functionality is being adversely affected by the bug-fixes. When the degree is high, the developers are informed about it. This metric is measured as

Number of bugs remaining per fix

Defect Density

This metric is defined as

$$\textit{Defect density} = \frac{\textit{Total number of defects found for a requirement}}{\textit{Number of test cases executed for that requirement}}$$

It indicates the average number of bugs found per test case in a specific functional area or requirement. If the defect density is high in a specific functionality, then there is some problem in the functionality due to the following reasons:

1. Complex functionality
2. Problem with design or implementation of that functionality
3. Inadequate resources are assigned to the functionality

This metric can also be utilized after the release of the product. The defect data should also be collected after the release of the product. For example, we can use the following metric:

Pre-ship defect density / Post-ship defect density

This metric gives the indication of how many defects remain in the software when it is released. Ideally, the number of pre-release defects found should increase and post-release defects should decrease.

Coverage Measures

Coverage goals can be planned, against which the actual ones can be measured. For white-box testing, a combination of following is recommended: degree of statement, branch, data flow, basis path, etc. coverage (planned, actual).

In this way, testers can use the following metric:

Actual degree of coverage / Planned degree of coverage

Similarly, for black-box testing, the following measures can be determined: (i) number of requirements or features to be tested, and (ii) number of equivalence classes identified.

In this way, testers can use the following metric:

Number of features or equivalence classes actually covered / Total number of features or equivalence classes.

These metrics will help in identifying the work to be done.

Tester Productivity Measures

Testers' productivity in unit time can also be measured and tracked. Managers can use these metrics for their team's productivity and build a confidence level for the on-date delivery of the product.

The following are some useful metrics in measuring the testers' productivity:

1. Time spent in test planning
2. Time spent in test case design
3. Time spent in test execution
4. Time spent in test reporting
5. Number of test cases developed
6. Number of test cases executed

Each of these metrics is planned and actual ones are measured and used for tracking the productivity.

Budget and Resource Monitoring Measures

The budget and other resources involved in a project are well-planned and must be tracked with the actual ones. Otherwise, a project may stop in between due to over-budget or fall behind its schedule.

Earned value tracking [108,109] given by Humphrey is the measure which is used to monitor the use of resources in testing. Test planners first estimate the total number of hours or cost involved in testing. Each testing activity is then assigned a value based on the estimated percentage of the total time or budget. It provides a relative value to each testing activity with respect to the entire testing effort. Then, this metric can be measured for the actual one and compared with the planned earned value.

For the planned earned values, we need the following measurement data:

1. Total estimated time or cost for overall testing effort.
2. Estimated time or cost for each testing activity.
3. Actual time or cost of each testing activity.

Then, the following metric is used to track the cost and time:

Estimated cost or time for testing activity / Actual cost or time of testing activity

Earned values are usually represented in a table form, as shown in Table 11.17.

Table 11.17 Earned values

Planned				Actual		
Testing Activity	Estimated Time (HRS)	Estimated Earned Value	Cumulative Earned Value	Date	Actual Earned value	Cumulative Earned Value

This table has two partitions: *planned values* and *actual values*. Each testing activity as well as their estimated hours for completion should be listed. The total hours for all the tasks are determined and the estimated earned value for each activity is then calculated based on their estimated percentage of the total time. This gives a relative value to each testing activity with respect to the entire testing effort. The estimated earned values are accumulated in the next column. When the testing effort is in progress, the date and actual earned value for each activity as well as their actual accumulated earned values.

Test Case Effectiveness Metric

This metric was defined by Chernak [110]. It shows how to determine whether a set of test cases is sufficiently effective in revealing defects. It is defined as

$$TCE = \frac{Number\ of\ defects\ found\ by\ the\ test\ cases}{Total\ number\ of\ defects} \times 100$$

In fact, a baseline value is selected for the TCE and is assigned for the project. When the TCE value is at or above the baseline, then the test cases have been found effective for testing. Otherwise, the test cases must be redesigned to increase the effectiveness.

SUMMARY

This chapter discusses the importance of testing effort measurements so that the largest phase of SDLC, i.e. testing, does not go beyond its schedule and budget. Therefore, testing metrics are designed to track and control all the testing activities at every phase of STLC. But before designing the testing metrics, we must know the attributes of testing to be measured, as we have seen in Chapter 10 on software measurement that attributes are necessary to design the metrics. We have recognized four major categories of attributes: progress, quality, cost, and size. All the attributes under these categories have been discussed in detail.

Various testing effort estimation models have also been discussed. These models consider various strategies to measure the effort made on testing activities. For example, Halstead model, function-point based model, test-point based model, etc. All these models provide metrics which can be used to measure the testing effort. Some of the models are very well-established in the industry. After covering the testing effort estimation model, finally, we discussed some of the well-known testing metrics.

Let us review the important concepts described in this chapter:

- Defect backlog is the number of defects that are outstanding and unresolved at one point of time with respect to the number of defects occuring.
- Defect-removal efficiency is a powerful metric for test-effectiveness, which is defined as the ratio of number of faults actually found in testing and the number of faults that could have been found in testing.

- Defect age is another metric that can be used to measure the test-effectiveness, which assigns a numerical value to the fault, depending on the phase in which it is discovered. Defect age is used in another metric called defect spoilage to measure the effectiveness of defect removal activities. Defect spoilage is calculated as

  Spoilage = Sum of (number of defects × defect age) / Total number of defects
- Spoilage is more suitable for measuring the long-term trend of test effectiveness. Generally, low values of defect spoilage mean more effective defect discovery processes.
- The relationship between code coverage and the number of test cases is described by the expression $C(x) = 1 - e^{-(p/N)*x}$ where $C(x)$ is the coverage after executing x number of test cases, N is the number of blocks in the program and p is the average number of blocks covered by a test case during the function test.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. The function point metric is used effectively for measuring the _____ of a software system.
   (a) effort
   (b) time
   (c) size
   (d) none of the above
2. As the data complexity increases, the probability of errors in module *m* _____
   (a) remains the same
   (b) increases
   (c) decreases
   (d) none of the above
3. It will be difficult to unit test a module with _____ system complexity.
   (a) higher
   (b) lower
   (c) none of the above
4. Integration and integration testing effort increases with the _____ in architectural complexity of the system.
   (a) decrease
   (b) increase
   (c) none of the above
5. Cyclomatic number should be less than or equal to _____.
   (a) 8
   (b) 9
   (c) 10
   (d) 12

6. Earned value tracking is the measure which is used to monitor the use of _____ in testing.
   (a) resources
   (b) modules
   (c) test cases
   (d) none of the above
7. Test point analysis is a technique to measure the _____ test effort estimation.
   (a) black-box
   (b) white-box
   (c) black-box and white-box
   (d) none of the above
8. Function point analysis is a technique to measure the _____ test effort estimation.
   (a) black-box
   (b) white-box
   (c) black-box and white-box
   (d) none of the above
9. Estimating an overall amount of work involved by documenting the parts of the software to be tested, is known as _____.
   (a) staff productivity
   (b) progress
   (c) scope of testing
   (d) none of the above
10. Which information is not shown on the test progress S curve?
   (a) planned number of test cases to be completed successfully by week
   (b) number of test cases attempted by week
   (c) number of test cases completed successfully by week
   (d) testing defect backlog
11. Test point analysis uses the _____ for calculating test points.
   (a) function points
   (b) use case points
   (c) number of test cases
   (d) number of testers
12. QC_{dw} is assigned the value _____ for each quality characteristic which can be tested statically using the checklist.
   (a) 13
   (b) 16
   (c) 17
   (d) 19

Review Questions

1. Consider the following program for finding the divisors of a positive integer:

```
main()
{
 int number, flag;
 int CheckPrime(int n);

 printf("Enter the value of positive integer");
 scanf("%d", &number);

 if(number == 1)
   printf("Divisor of %d is %d", number, number);
 else
 {
   flag = CheckPrime(number);
   if (flag)
   {
           printf(" %d is a prime number", number);
           printf(" Divisors are 1 and %d", number);
   }
 }
}

int CheckPrime(int n)
{
 void divisors(int n);
 int squareroot, x;

 squareroot = sqrt(n);
 x = 2;
 while(x<=squareroot)
 {
   if(number == 0)
           break;
   x++;
 }
 if(x > squareroot)
   return(1);
 else
 {
   printf("%d is a composite number", number);
   printf("Divisors are");
   divisors(number);
   return(0);
 }
}
```

```
void divisors(int n)
{
 int x, mid;
 mid = n/2;
 for(x = 1; x<=mid; x++)
 {
   if (n % d == 0)
   printf("%d", x);
 }
 printf("%d", n);
}
```

(a) Calculate the following for each module in the program:
   (i) Fan-in
   (ii) Fan-out
   (iii) Structural complexity
   (iv) Data complexity
   (v) System complexity
   (vi) IFC

(b) Which module is difficult to unit test?

2. Consider a project with the following distribution of data and calculate its defect spoilage.

SDLC Phase	No. of defects	Defect age
Requirement Specs.	34	2
HLD	25	4
LLD	17	5
Coding	10	6

3. The module implementation details of a software project are given below:

Module	Unique operators	Unique operands	Total operators	Total operands
A	23	12	43	37
B	34	12	56	34
C	12	23	54	41

Calculate Halstead effort for all the modules and the percentage of overall testing effort to be allocated to each module.

4. There are 1200 estimated function points in a project. Calculate the total number of test cases in the system and the number of test cases in acceptance testing. Also calculate the defect density (number of total defects is 236) and test case coverage.

5. Consider the table in Question 2 and calculate the defect density at each phase considering FP = 980 for the system.

6. What are the consequences if the cyclomatic complexity of a system is greater than 10? As a designer of the system, what strategy will you choose to handle it?
7. In a software project, 2050 test cases were planned, out of which 1980 were executed till date. During the execution of these test cases, bugs were found with the following details:

Bug reference ID	Bug opening date	Bug closing date
M01-01	09-10-2007	10-10-2007
M01-02	12-10-2007	12-10-2007
M01-03	11-10-2007	14-10-2007
M02-01	13-10-2007	14-10-2007
M02-02	13-10-2007	15-10-2007
M02-03	14-10-2007	16-10-2007
M02-04	15-10-2007	15-10-2007
M02-05	17-10-2007	20-10-2007
M02-06	17-10-2007	19-10-2007

Compute the following for this project:

(a) Test procedure execution status
(b) Turnaround time for a defect to be corrected
(c) Bug trend analysis for defects per day that can be fixed
(d) Defect density for requirement number M01 and M02
(e) TCE

8. What is the importance of software attributes in measurement?
9. Explain the different categories of testing attributes.
10. Visit a CMM level 5 software industry. Survey the various testing metrics being used there. List all of them and mark their importance in STLC.
11. Obtain the test plan of a project and evaluate it on the basis of rubrics discussed in this chapter.
12. What is testing defect backlog?
13. What are the problems in measuring the staff productivity attribute?
14. Define and discuss the purpose of defect removal efficiency, defect age, and defect spoilage.
15. The historical record of some similar type of projects are shown below:

Number of test procedures (NTP)	Number of person-hours consumed for testing (PH)
866	4500
870	4512
956	4578
903	4520
790	4460

If a new project of similar types is to be developed for which the number of test procedures are 1245, calculate the number of testers required if the total period of testing scheduled is 2050 hours.

16. Calculate the total test points for the functions of a software. The specifications of all the modules are given in the table below.

Function	Specification
M01	Function points = 234, Ratings for all FDC_w factors are high. Uniformity factor =1, rating for all QC_{dw} are 'important' and for QC_{sw}, two static qualities are considered.
M02	Functions points = 340, Ratings for all FDC_w factors are normal. Uniformity factor = 1, rating for all QC_{dw} are 'very important' and for QC_{sw}, three static qualities are considered.
M03	Functions points = 450, Ratings for all FDC_w factors are high. Uniformity factor = 1, rating for all QC_{dw} are 'important' and for QC_{sw}, two static qualities are considered.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter 12

Efficient Test Suite Management

OBJECTIVES

After reading this chapter, you should be able to understand:

- Why the test suite grows as the software evolves
- Why is it required to minimize the number of test cases
- Factors to be considered while minimizing a test suite
- Definition of test suite minimization problem
- Test suite prioritization as a method to reduce test cases
- Prioritization techniques

Software testing is a continuous process that takes place throughout the life cycle of a project. Test cases in an existing test suite can often be used to test a modified program. However, if the test suite is inadequate for retesting, new test cases may be developed and added to the test suite. Thus, the size of a test suite grows as the software evolves. Due to resource constraints, it is important to prioritize the execution of test cases so as to increase the chances of early detection of faults. A reduction in the size of the test suite decreases both the overhead of maintaining the test suite and the number of test cases that must be rerun after changes are made to the software. This chapter discusses the techniques of minimizing a test suite.

12.1 Why Does a Test Suite Grow?

A testing criterion is a rule or collection of rules that imposes requirements on a set of test cases. Test engineers measure the extent to which a criterion is satisfied in terms of coverage; a test set achieves 100% coverage if it completely satisfies the criterion. *Coverage* is measured in terms of the requirements that are imposed; *partial coverage* is defined as the percentage of requirements that are satisfied.

Test requirements are specific things that must be satisfied or covered; e.g. for statement coverage, each statement is a requirement; in mutation, each mutant is a requirement; and in data flow testing, each DU pair is a requirement.

Coverage criteria are used as a stopping point to decide when a program is sufficiently tested. In this case, additional tests are added until the test suite has achieved a specified coverage level according to a specific adequacy criterion. For example, to achieve statement coverage adequacy for a program, one would add additional test cases to the test suite until each statement in that program is executed by at least one test case.

Test suites can also be reused later as the software evolves. Such test suite reuse, in the form of regression testing, is pervasive in the software industry. These tests account for as much as one half of the cost of software maintenance. Running all the test cases in a test suite, however, can require a lot of effort.

There may be unnecessary test cases in the test suite including both obsolete and redundant test cases. A change in a program causes a test case to become obsolete by removing the reason for the test case's inclusion in the test suite. A test case is redundant if other test cases in the test suite provide the same coverage of the program. Thus, due to obsolete and redundant test cases, the size of a test suite continues to grow unnecessarily.

12.2 Minimizing the Test Suite and its Benefits

A test suite can sometimes grow to an extent that it is nearly impossible to execute. In this case, it becomes necessary to minimize the test cases such that they are executed for maximum coverage of the software. Following are the reasons why minimization is important:

- Release date of the product is near.
- Limited staff to execute all the test cases.
- Limited test equipments or unavailability of testing tools.

When test suites are run repeatedly for every change in the program, it is of enormous advantage to have as small a set of test cases as possible. Minimizing a test suite has the following benefits:

- Sometimes, as the test suite grows, it can become prohibitively expensive to execute on new versions of the program. These test suites will often contain test cases that are no longer needed to satisfy the coverage criteria, because they are now obsolete or redundant. Through minimization, these redundant test cases will be eliminated.
- The sizes of test sets have a direct bearing on the cost of the project. Test suite minimization techniques lower costs by reducing a test suite to a minimal subset.

- Reducing the size of a test suite decreases both the overhead of maintaining the test suite and the number of test cases that must be rerun after changes are made to the software, thereby reducing the cost of regression testing.

Thus, it is of great practical advantage to reduce the size of test cases.

12.3 Defining Test Suite Minimization Problem

Harrold *et al.* [51] have defined the problem of minimizing the test suite as given below.

Given A test suite TS; a set of test case requirements $r_1, r_2, \ldots, r_n$ that must be satisfied to provide the desired testing coverage of the program; and subsets of TS, $T_1, T_2, \ldots, T_n$, one associated with each of the r_i's such that any one of the test cases t_j belonging to T_i can be used to test r_i.

Problem Find a representative set of test cases from *TS* that satisfies all the r_i's.

The r_i's can represent either all the test case requirements of a program or those requirements related to program modifications. A representative set of test cases that satisfies the r_i's must contain at least one test case from each T_i. Such a set is called a *hitting set* of the group of sets, $T_1, T_2, \ldots T_n$. Maximum reduction is achieved by finding the smallest representative of test cases. However, this subset of the test suite is the minimum cardinality hitting set of the T_i's and the problem of finding the minimum cardinality hitting set is *NP*-complete. Thus, minimization techniques resort to heuristics.

12.4 Test Suite Prioritization

The reduction process can be best understood if the cases in a test suite are prioritized in some order. The purpose of prioritization is to reduce the set of test cases based on some rational, non-arbitrary criteria, while aiming to select the most appropriate tests. For example, the following priority categories can be determined for the test cases:

Priority 1 *The test cases must be executed*, otherwise there may be worse consequences after the release of the product. For example, if the test cases for this category are not executed, then critical bugs may appear.

Priority 2 *The test cases may be executed, if time permits.*

Priority 3 *The test case is not important prior to the current release.* It may be tested shortly after the release of the current version of the software.

Priority 4 *The test case is never important, as its impact is nearly negligible.*

In the prioritization scheme, the main guideline is to ensure that low-priority test cases do not cause any severe impact on the software. There may be several goals of prioritization. These goals can become the basis for prioritizing the test cases. Some of them are discussed here:

- Testers or customers may want to get some critical features tested and presented in the first version of the software. Thus, the important features become the criteria for prioritizing the test cases. But the consequences of not testing some low-priority features must be checked. Therefore, risk factor should be analysed for every feature in consideration.
- Prioritization can be on the basis of the functionality advertised in the market. It becomes important to test those functionalities on a priority basis, which the company has promised to its customers.
- The rate of fault detection of a test suite can reveal the likelihood of faults earlier.
- Increase the coverage of coverable code in the system under test at a faster rate, allowing a code coverage criterion to be met earlier in the test process.
- Increase the rate at which high-risk faults are detected by a test suite, thus locating such faults earlier in the testing process.
- Increase the likelihood of revealing faults related to specific code changes, earlier in the regression testing process.

12.5 Types of Test Case Prioritization

General Test Case Prioritization

In this prioritization, we prioritize the test cases that will be useful over a succession of subsequent modified versions of P, without any knowledge of the modified versions. Thus, a general test case prioritization can be performed following the release of a program version during off-peak hours, and the cost of performing the prioritization is amortized over the subsequent releases.

Version-Specific Test Case Prioritization

Here, we prioritize the test cases such that they will be useful on a specific version P' of P. Version-specific prioritization is performed after a set of changes

have been made to P and prior to regression testing of P', with the knowledge of the changes that have been made.

12.6 Prioritization Techniques

Prioritization techniques schedule the execution of test cases in an order that attempts to increase their effectiveness at meeting some performance goal. Therefore, given any prioritization goal, various prioritization techniques may be applied to a test suite with the aim of meeting that goal. For example, in an attempt to increase the rate of fault-detection of test suites, we might prioritize test cases in terms of the extent to which they execute modules that, measured historically, tend to fail. Alternatively, we might prioritize the test cases in terms of their increasing cost-per-coverage of code components, or in terms of their increasing cost-per-coverage of features listed in a requirement specification. In any case, the intent behind the choice of a prioritization technique is to increase the likelihood that the prioritized test suite can better meet the goal than an *ad hoc* or random ordering of test cases.

Prioritization can be done at two levels, as discussed below.

Prioritization for regression test suite This category prioritizes the test suite of regression testing. Since regression testing is performed whenever there is a change in the software, we need to identify the test cases corresponding to modified and affected modules.

Prioritization for system test suite This category prioritizes the test suite of system testing. Here, the consideration is not the change in the modules. The test cases of system testing are prioritized based on several criteria: risk analysis, user feedback, fault-detection rate, etc.

12.6.1 Coverage-based Test Case Prioritization

This type of prioritization [58] is based on the coverage of codes, such as statement coverage, branch coverage, etc. and the fault exposing capability of the test cases. Test cases are ordered based on their coverage. For example, count the number of statements covered by the test cases. The test case that covers the highest number of statements will be executed first. Some of the techniques are discussed below.

Total Statement Coverage Prioritization

This prioritization orders the test cases based on the total number of statements covered. It counts the number of statements covered by the test cases

and orders them in a descending order. If multiple test cases cover the same number of statements, then a random order may be used.

For example, if T_1 covers 5 statements, T_2 covers 3, and T_3 covers 12 statements; then according to this prioritization, the order will be T_3, T_1, T_2.

Additional Statement Coverage Prioritization

Total statement coverage prioritization schedules the test cases based on the total statements covered. However, it will be useful if it can execute those statements as well that have not been covered yet. Additional statement coverage prioritization iteratively selects a test case T_1, that yields the greatest statement coverage, then selects a test case which covers a statement uncovered by T_1. Repeat this process until all statements covered by at least one test case have been covered.

For example, if we consider Table 12.1, according to total statement coverage criteria, the order is 2,1,3. But additional statement coverage selects test case 2 first and next, it selects test case 3, as it covers statement 4 which has not been covered by test case 2. Thus, the order according to addition coverage criteria is 2,3,1.

Table 12.1 Statement coverage

Statement	Statement Coverage		
	Test case 1	Test case 2	Test case 3
1	X	X	X
2	X	X	X
3		X	X
4			X
5			
6		X	
7	X	X	
8	X	X	
9	X	X	

Total Branch Coverage Prioritization

In this prioritization, the criterion to order is to consider condition branches in a program instead of statements. Thus, it is the coverage of each possible outcome of a condition in a predicate. The test case which will cover maximum branch outcomes will be ordered first. For example, see Table 12.2. Here, the order will be 1, 2, 3.

Table 12.2 Branch coverage

Branch Statements	Branch Coverage		
	Test case 1	Test case 2	Test case 3
Entry to while	X	X	X
2-true	X	X	X
2-false	X		
3-true		X	
3-false	X		

Additional Branch Coverage Prioritization

Here, the idea is the same as in additional statement coverage of first selecting the test case with the maximum coverage of branch outcomes and then, selecting the test case which covers the branch outcome not covered by the previous one.

Total Fault-Exposing-Potential (FEP) Prioritization

Statement and branch coverage prioritization ignore a fact about test cases and faults:

- Some bugs/faults are more easily uncovered than other faults.
- Some test cases have the proficiency to uncover particular bugs as compared to other test cases.

Thus, the ability of a test case to expose a fault is called the *fault exposing potential*. It depends not only on whether test cases cover a faulty statement, but also on the probability that a fault in that statement will also cause a failure for that test case.

To obtain an approximation of the FEP of a test case, an approach was adopted using mutation analysis [137,138]. This approach is discussed below.

Given a program P and a test suite T,

- First, create a set of mutants $N = \{n_1, n_2, \ldots n_m\}$ for P, noting which statement s_j in P contains each mutant.
- Next, for each test case $t_i T$, execute each mutant version n_k of P on t_i, noting whether t_i kills that mutant.
- Having collected this information for every test case and mutant, consider each test case t_i, and each statement s_j in P, and calculate $FEP(s,t)$ of t_i on s_j as the ratio:

  Mutants of s_j killed / Total number of mutants of s_j

  If t_i does not execute s_j, this ratio is zero.

To perform total FEP prioritization, given these (FEP)(s,t) values, calculate an *award value* for each test case t_iT, by summing the (FEP)(s_j, t_i) values for all statements s_j in P. Given these award values, we prioritize the test cases by sorting them in order of descending award value.

Table 12.3 shows FEP estimates of a program. Total FEP prioritization outputs the test case order as (2, 3, 1).

Table 12.3 FEP estimates

Statement	FEP(s,t) values		
	Test case 1	Test case 2	Test case 3
1	0.5	0.5	0.3
2	0.4	0.5	0.4
3		0.01	0.4
4		1.3	
5			
6	0.3		
7	0.6		0.1
8		0.8	0.2
9			0.6
Award values	1.8	3.11	2.0

12.6.2 Risk-based Prioritization

Risk-based prioritization [22] is a well-defined process that prioritizes modules for testing. It uses risk analysis to highlight potential problem areas, whose failures have adverse consequences. The testers use this risk analysis to select the most crucial tests. Thus, risk-based technique is to prioritize the test cases based on some potential problems which may occur during the project.

- **Probability of occurrence/fault likelihood** It indicates the probability of occurrence of a problem.
- **Severity of impact/failure impact** If the problem has occurred, how much impact does it have on the software.

  Risk analysis uses these two components by first listing the potential problems and then, assigning a probability and severity value for each identified problem, as shown in Table 12.4. By ranking the results in this table in the form of risk exposure, testers can identify the potential

problems against which the software needs to be tested and executed first. For example, the problems in the given table can be prioritized in the order of P_5, P_4, P_2, P_3, P_1.

A risk analysis table consists of the following columns:

- **Problem ID** A unique identifier to facilitate referring to a risk factor.
- **Potential problem** Brief description of the problem.
- **Uncertainty factor** It is the probability of occurrence of the problem. Probability values are on a scale of 1 (low) to 10 (high).
- **Severity of impact** Severity values on a scale of 1 (low) to 10 (high).
- **Risk exposure** Product of probability of occurrence and severity of impact.

Table 12.4 Risk analysis table

Problem ID	Potential Problem	Uncertainty Factor	Risk Impact	Risk Exposure
P_1	Specification ambiguity	2	3	6
P_2	Interface problems	5	6	30
P_3	File corruption	6	4	24
P_4	Databases not synchronized	8	7	56
P_5	Unavailability of modules for integration	9	10	90

12.6.3 Prioritization Based on Operational Profiles

This is not a prioritization in true sense, but the system is developed in such a way that only useful test cases are designed. So there is no question of prioritization. In this approach, the test planning is done based on the operation profiles [128, 129] of the important functions which are of use to the customer. An operational profile is a set of tasks performed by the system and their probabilities of occurrence. After estimating the operational profiles, testers decide the total number of test cases, keeping in view the costs and resource constraints.

12.6.4 Prioritization using Relevant Slices

During regression testing, the modified program is executed on all existing regression test cases to check that it still works the same way as the original

program, except where a change is expected. But re-running the test suite for every change in the software makes regression testing a time-consuming process. If we can find the portion of the software which has been affected with the change in software, then we can prioritize the test cases based on this information. This is called the *slicing technique* [130,131,132]. The various definitions related to this technique have been defined in literature. Let us discuss them.

Execution Slice

The set of statements executed under a test case is called the *execution slice* of the program. For example, see the following program (Fig. 12.1):

```
Begin
S1: read (basic, empid);
S2: gross = 0;
S3: if (basic > 5000 || empid > 0)
         {
S4:       da = (basic*30)/100;
S5:       gross = basic + da;
         }
S6:  else
         {
S7:       da = (basic*15)/100;
S8:       gross = basic + da;
          }
S9: print (gross, empid);
End
```

Figure 12.1 Example program for execution slice

Table 12.5 shows the test cases for the given program.

Table 12.5 Test cases

Test Case	Basic	Empid	Gross	Empid
T1	8000	100	10400	100
T2	2000	20	2300	20
T3	10000	0	13000	0

$T1$ and $T2$ produce correct results. On the other hand, $T3$ produces an incorrect result. Syntactically, it is correct, but an employee with the empid '0' will not get any salary, even if his basic salary is read as input. So it has to be modified.

Suppose $S3$ is modified as [if (basic>5000 && empid>0)]. It means for $T1$, $T2$, and $T3$, the program would be rerun to validate whether the change in $S3$ has introduced new bugs or not. But if there is a change in $S7$ statement [da = (basic*25)/100; instead of da = (basic*15)/100;], then only $T2$ will be rerun. So in the execution slice, we will have less number of statements. The execution slice is highlighted in the given code segment.

```
Begin
  S1: read (basic, empid);
  S2: gross=0;
  S3: if(basic > 5000 || empid > 0)
        {
  S4:     da = (basic*30)/100;
  S5:     gross = basic + da;
        }
  S6: else
        {
  S7:     da = (basic*15)/100;
  S8:     gross = basic + da;
        }
  S9: print(gross, empid);
End
```

Instead of computing an execution slice at the statement level, we may also compute it at the function or the module level. That is, instead of determining which program statements are executed under each test case, we may simply determine which functions or modules are invoked under each test case. Then, we need to execute the new program on a test case only if a modified function or module was invoked during the original program's execution on that test case.

Dynamic Slice

The set of statements executed under a test case having an effect on the program output is called the *dynamic slice* of the program with respect to the output variables. For example, see the following program (Fig. 12.2).

```
Begin
  S1: read (a,b);
  S2: sum=0;
  S2.1: I=0;
  S3: if (a==0)
```

```
        {
S4:        print(b);
S5:        sum+=b;
        }
S6: else if(b==0)
        {
S7:        print(a);
S8:        sum+=a;
        }
S9: else
        {
S10:       sum=a+b+sum;
S10.1      I=25;
S10.2      print(I);
        }
S11:endif
S12:print(sum);
End
```

Figure 12.2 Example program for dynamic slice

Table 12.6 shows the test cases for the given program.

Table 12.6 Test cases

Test Case	a	b	sum
T1	0	4	4
T2	67	0	67
T3	23	23	46

$T1$, $T2$, and $T3$ will run correctly but if some modification is done in $S10.1$ [say I = 50], then this change will not affect the output variable. Therefore, there is no need to rerun any of the test cases. On the other hand, if $S10$ is changed [say, sum=a*b+sum], then this change will affect the output variable 'sum', so there is a need to rerun $T3$. The dynamic slice is highlighted in the code segment.

```
Begin
S1:     read (a,b);
S2:     sum = 0;
S2.1:   I=0;
S3:     if (a==0)
        {
S4:     print(b);
S5:     sum+=b;
        }
```

```
    S6: elseif(b==0)
            {
    S7:     print(a);
    S8:     sum+=a;
                }
    S9: else
            {
    S10:    sum=a+b+sum;
    S10.1   I=25;
    S10.2   print(I);
            }
    S11: endif
    S12: print(sum);
End
```

Relevant Slice

The set of statements that were executed under a test case and did not affect the output, but have the potential to affect the output produced by a test case, is known as the *relevant slice* of the program. It contains the dynamic slice and in addition, includes those statements which, if corrected, may modify the variables at which the program failure has manifested. For example, consider the example of Fig. 12.2, statements $S3$ and $S6$ have the potential to affect the output, if modified.

If there is a change in any statement in the relevant slice, we need to rerun the modified software on only those test cases whose relevant slices contain a modified statement. Thus, on the basis of relevant slices, we can prioritize the test cases. This technique is helpful for prioritizing the regression test suite which saves time and effort for regression testing.

Jeffrey and Gupta [139] enhanced the approach of relevant slicing and stated: 'If a modification in the program has to affect the output of a test case in the regression test suite, it must affect some computation in the relevant slice of the output for that test case'. Thus, they applied the heuristic for prioritizing test cases such that the test case with larger number of statements must get higher weight and will get priority for execution.

12.6.5 Prioritization Based on Requirements

This technique is used for prioritizing the system test cases. The system test cases also become too large in number, as this testing is performed on many grounds. Since system test cases are largely dependent on the requirements, the requirements can be analysed to prioritize the test cases.

This technique does not consider all the requirements on the same level. Some requirements are more important as compared to others. Thus, the test cases corresponding to important and critical requirements are given more weight as compared to others, and these test cases having more weight are executed earlier.

Hema Srikanth *et al.* [136] have applied requirement engineering approach for prioritizing the system test cases. It is known as PORT (prioritization of requirements for test). They have considered the following four factors for analysing and measuring the criticality of requirements:

Customer-assigned priority of requirements Based on priority, the customer assigns a weight (on a scale of 1 to 10) to each requirement. Higher the number, higher is the priority of the requirement.

Requirement volatility This is a rating based on the frequency of change of a requirement. The requirement with a higher change frequency is assigned a higher weight as compared to the stable requirements.

Developer-perceived implementation complexity All the requirements are not equal on a implementation level. The developer gives more weight to a requirement which he thinks is more difficult to implement.

Fault proneness of requirements This factor is identified based on the previous versions of system. If a requirement in an earlier version of the system has more bugs, i.e. it is error-prone, then this requirement in the current version is given more weight. This factor cannot be considered for a new software.

Based on these four factor values, a *prioritization factor value* (PFV) is computed as given below. PFV is then used to produce a prioritized list of system test cases.

$$\mathbf{PFV}_i = \sum(\mathbf{FV}_{ij} \times \mathbf{FW}_j)$$

where FV_{ij} = Factor value is the value of factor j corresponding to requirement i
FW_j = Factor weight is the weight given to factor j

12.7 Measuring the Effectiveness of a Prioritized Test Suite

When a prioritized test suite is prepared, how will we check its effectiveness? We need one metric which can tell us the effectiveness of one prioritized test suite. For this purpose, the rate of fault-detection criterion can be taken. Elbaum *et al.* [133,134] developed APFD (average percentage of faults detected) metric that measures the weighted average of the percentage of faults detected

during the execution of a test suite. Its value ranges from 0 to 100, where a higher value means a faster fault-detection rate. Thus, APFD is a metric to detect how quickly a test suite identifies the faults. If we plot percentage of test suite run on the x-axis and percentage of faults detected on the y-axis, then the area under the curve is the APFD value, as shown in Fig. 12.3.

Figure 12.3 Calculating APFD

APFD is calculated as given below.

$$\text{APFD} = 1 - ((\text{TF}_1 + \text{TF}_2 + \ldots\ldots\ldots + \text{TF}_m)/nm) + 1/2n$$

where TF_i is the position of the first test in test suite T that exposes fault i
m is the total number of faults exposed in the system or module under T
n is the total number of test cases in T

Example 12.1

Consider a program with 10 faults and a test suite of 10 test cases, as shown in Table 12.7.

Table 12.7 Test case fault exposure

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
F1					X			X		
F2		X	X	X		X				
F3	X	X	X	X			X	X		
F4						X				X
F5	X		X		X	X		X		X
F6					X	X	X		X	
F7									X	
F8		X		X		X			X	X
F9	X									X
F10			X	X					X	X

If we consider the order of test suite as (T1, T2, T3, T4, T5, T6, T7, T8, T9, T10), then calculate the APFD for this program.

Solution

$$\text{APFD} = 1 - ((5 + 2 + 1 + 6 + 1 + 5 + 9 + 2 + 1 + 3)/10*10) + \frac{1}{2 \times 10}$$

$$= 0.65 + 0.05$$
$$= 0.7$$

All the bugs detected are not of the same level of severity. One bug may be more critical as compared to others. Moreover, the cost of executing the test cases also differs. One test case may take more time as compared to others. Thus, APFD does not consider the severity level of the bugs and the cost of executing the test cases in a test suite. Elbaum *et al.* [135] modified their APFD metric and considered these two factors to form a new metric which is known as *cost-cognizant* APFD and denoted as APFD_c. In APFD_c, the total cost incurred in all the test cases is represented on x-axis and the total fault severity detected is taken on y-axis. Thus, it measures the unit of fault severity detected per unit test case cost.

SUMMARY

The size of a test suite increases as the software evolves. We have also seen this in regression testing. This chapter separately deals with this problem because test suite size is a general problem, not restricted to regression testing. In this chapter, we discuss the benefits of reducing the number of test cases along with the general definition of test suite minimization problem.

Test case prioritization technique has become very popular and effective today in minimizing the test suite. Prioritization can be done at various levels like statement coverage, branch coverage, or fault exposing potential. The prioritization technique and its various methods have also been discussed here.

Let us review the important concepts described in this chapter:

- The purpose of prioritization is to reduce the set of test cases based on some rational, non-arbitrary criteria, while aiming to select the most appropriate tests.
- General test case prioritization technique prioritizes the test cases that will be useful over a succession of subsequent modified versions of *P,* without any knowledge of modified versions.
- Version-specific test case prioritization prioritizes the test cases such that they will be useful on a specific version *P'* of *P*.
- Risk-based prioritization technique is to prioritize the test cases based on some potential problems which may occur during the project.
- The set of statements executed under a test case is called the execution slice of the program.

- The set of statements executed under a test case having an effect on the program output is called the dynamic slice of the program with respect to the output variables.
- The set of statements that were executed under a test case and did not affect the output, but have the potential to affect the output produced by a test case is known as the relevant slice of a program.
- Prioritization factor value (PFV) is used to produce a prioritized list of system test cases as given below:

$$PFV_i = \sum (FV_{ij} \times FW_j)$$

where FV = Factor value is the value of factor *j* corresponding to requirement *i*

FW = Factor weight is the weight given to factor *j*

- APFD (average percentage of faults detected) is the metric that measures the weighted average of the percentage of faults detected during the execution of a test suite. Its value ranges from 0 to 100, where a higher value means a faster fault-detection rate.

$$APFD = 1 - ((TF_1 + TF_2 + \ldots\ldots\ldots + TF_m) / nm) + 1/2n$$

where TF_i is the position of the first test in test suite *T* that exposes fault *i*

m is the total number of faults exposed in the system or module under *T*

n is the total number of test cases in *T*

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. If the test suite is inadequate for retesting, then _____.
   (a) new test cases may be developed and added to the test suite
   (b) existing test suite should be modified accordingly
   (c) old test suite should be discarded and altogether new test suite should be developed
   (d) none of the above
2. The size of a test suite _____ as the software evolves.
   (a) decreases
   (b) increases
   (c) remains same
   (d) none of the above
3. Coverage is measured in terms of the _____ that are imposed.
   (a) requirements
   (b) design
   (c) test cases
   (d) none of the above

4. In the prioritization scheme, the main guideline is to ensure that _____ priority test cases do not cause any severe impacts on the software.
   (a) high
   (b) low
   (c) medium
   (d) none of the above
5. Automatic test generation often results in _____ test sets.
   (a) larger
   (b) smaller
   (c) medium size
   (d) none of the above
6. The set of statements executed under a test case, having an effect on the program output under that test case is called _____.
   (a) execution slice
   (b) dynamic slice
   (c) relevant slice
   (d) none of the above
7. The set of statements executed under a test case is called _____.
   (a) execution slice
   (b) dynamic slice
   (c) relevant slice
   (d) none of the above
8. The set of statements that were executed under a test case and did not affect the output, but have the potential to affect the output produced by a test case is known as _____.
   (a) execution slice
   (b) dynamic slice
   (c) relevant slice
   (d) none of the above
9. Which one is true?
   (a) APFD = $1 + ((TF_1+TF_2+\ldots\ldots\ldots+ TF_m) / nm) + 1/2n$
   (b) APFD = $1- ((TF_1+TF_2+\ldots\ldots\ldots+ TF_m) / nm) + 1/3n$
   (c) APFD = $1- ((TF_1+TF_2+\ldots\ldots\ldots+ TF_m) / nm) + 1/2n$
   (d) none of the above

Review Questions

1. What is the need for minimizing the test cases in a project?
2. Develop a priority category scheme for the test cases to be executed in a project that deals with all kinds of priorities set in that project.

3. Identify some potential problems in a project. Mark them on a scale of 1 to 10 for uncertainty factor and risk impact. Prepare its risk table.
4. Explain the following with example:
   (a) Total statement coverage prioritization
   (b) Additional statement coverage prioritization
   (c) Total branch coverage prioritization
   (d) Additional branch coverage prioritization
   (e) Total fault-exposing-potential (FEP) prioritization
5. Take a module from a project and its test cases. Identify execution, dynamic, and relevant slices in this module.
6. PORT depends on four factors. Are these factors sufficient for prioritization? If no, identify some more factors.
7. What is the use of PFV?
8. Search the literature on prioritization, based on operational profiles, and implement this technique in a project.
9. What is the difference between APFD and $APFD_c$?
10. Consider Example 12.1 and calculate its APFD with the following order of test suite:
    (a) (T2,T1,T3,T4,T5,T6,T7,T8,T9,T10)
    (b) (T1,T2,T5,T4,T3,T6,T9,T8,T7,T10)

    What is the effect of changing the order of test cases?

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Part 4 Quality Management

CHAPTERS

Chapter 13:
Software Quality Management

Chapter 14:
Testing Process Maturity Models

One of the major testing goals is to ensure software quality. We define testing as a critical element of software quality. This is where the story of software testing begins. We have mentioned this in Part I. In Part IV, we discuss in detail the factors that contribute to software quality, the role of testing in quality, and how to manage quality.

As software testing aims to get better quality software, the quality of the testing process also needs to be measured. There have been various quality models for SDLC, but none of them take the testing process into consideration. Therefore, testing process should also be measured for its maturity level with new models. The various models for testing the process maturity have been discussed in this part.

This part will make ground for the following concepts:

- Quality control
- Quality assurance
- Quality management
- Quality factors
- Methods of quality management
- Test process maturity models

Chapter

13

Software Quality Management

We use the term *quality* very often in our daily life. We expect a certain level of quality from a product. But how do we define quality? Crosby [39] defines quality as *conformance to requirements*. This means that the product should be developed according to the pre-specified requirements. If the requirements are misunderstood and the product is developed with incorrect requirements, then the product lacks quality.

Juran & Gryna [40] defines quality as *fitness for use*. It means whether the product is actually usable by the user or not. If the user is satisfied with the product, it is of good quality.

This shows that quality is not a single idea; rather it is a multi-dimensional concept, as shown in Fig. 13.1.

On the basis of the multi-dimensional concept, quality can be defined as 'the degree to which a product or service possesses a desired combination of attributes'.

OBJECTIVES

After reading this chapter, you should be able to understand:

- Software Quality is a multi-dimensional concept
- Difference between product quality and process quality
- Costs incurred to maintain quality
- Benefits of investment on quality
- Difference between quality control, quality assurance, and quality management
- Difference between quality management and project management
- Various factors of quality
- Types of quality management: procedural and quantitative
- Various quality metrics

Figure 13.1 Quality as multi-dimensional concept

Similarly, Pressman [7] defines quality as *an attribute of an item; quality refers to measurable characteristics—things we are able to compare to known standards such as length, colour, electrical properties, etc.*

13.1 Software Quality

We know that software is not similar to any physical product, so the concept of software quality also differs from physical products. Software quality is not easily definable. Software has many possible quality characteristics. In the narrowest sense, software quality revolves around defects. It is commonly recognized as lack of bugs in the software. This definition is the same as the classical definition of quality–'conformance to requirements'. If a software has functional bugs, then it may be possible that the software is not able to perform the functions. Thus, software quality may be defined in the form of delivered defect density or defect rate, i.e. the number of defects per unit size (e.g. LOC, FP, or other unit).

However, as discussed above, software quality is also a multi-dimensional concept rather than a single concept. In a software project, every member of the project team is responsible for the quality of the product. In fact, quality, along with cost and schedule is an important activity in a software project, which concerns everyone in the team. The ultimate goal is that the final product should have as few defects as possible, resulting in high quality software.

13.2 Broadening the Concept of Quality

Till now, software has been considered a product and in quality sense, we expect all its attributes to improve. But the definition of quality is not complete at this point. The concept of quality can be extended to the process to be adopted for development of the product. In this sense, there are two views of quality.

Product Quality

Quality measures are taken to improve the end-product.

Process Quality

From customers' requirements to the delivery of end-product, the development process to be adopted for software development is complex. If the quality of development process is not good enough, then it will certainly affect the quality of the end-product. If each stage of the development process is developed according to the requirement of its intermediate user, then the final end-product will be of high quality. Therefore, the quality of development process directly affects the quality of delivered product, as shown in Fig. 13.2.

Figure 13.2 Process affects quality

From the customer's point of view, product quality is according to the customer, while process quality is according to the intermediate user of the next stage. For example, the design person will use SRS. But all the intermediate stages of development process are being developed for a high-quality end-product.

Process quality is particularly useful in software development, as it is difficult to measure software attributes, such as maintainability, without using the software for a long period. Quality improvement focuses on identifying good quality products and examining the processes so that they may be applied across a range of projects.

To improve the quality of processes and products, the following activities must be carried out:

- Choose the development process best suited for the project.
- Select and replay specific methods and approaches within the selected process.
- Adopt proper tools and technologies.
- Prepare quality metrics for products based on their desired attributes and implement software metrics program to ensure that the development process is under control and making progress.

13.3 Quality Cost

Quality of a software is maintained with efforts in both products and processes, as shown above. If the efforts in the form of quality evaluation programs are not implemented, then the desired-level quality cannot be achieved. According to John Ruskin, *quality is never by an accident, it is always the result of an intelligent effort.*

But implementation of those quality-related procedures and practices is expensive and incurs cost. According to Pressman and Humphrey [7,41], the cost of quality is decomposed into three major categories:

- **Prevention costs** are related with activities that identify the cause of defects and those actions that are taken to prevent them, e.g. quality planning, formal technical reviews, test and laboratory equipments, training, defect causal analysis and prevention.
- **Appraisal costs** include the costs of evaluating the quality of software products at some level, e.g. testing, in-process and inter-process inspection by implementing software metrics programs (SMP), equipment calibrations, and maintenance.
- **Failure costs** include the costs to analyse and remove the failures. Since failures can occur at the development site as well as at the customer's site, failure costs can be divided into the following parts:
  - *External failure costs* are associated with failures that appear at the customer's site when the product has been released, e.g. regression testing corresponding to any bug, listening and resolving the customers' problems in case of any complaint from the customer, product return and replacement, etc.
  - *Internal failure costs* are those costs that appear on the developer's site prior to the release of the product, e.g. regression testing, isolating the bug and repairing, analysing the criticality of bug, etc.

As discussed earlier, the cost of finding and debugging a bug increases tenfold as we move further in the development stages of SDLC. Similarly, if we are not implementing prevention and appraisal activities in the early stages, then a high cost of failure detection and debugging has to be paid for internal and external failures.

13.4 Benefits of Investment on Quality

Despite the costs incurred in prevention and appraisal activities, benefits of these activities greatly outweigh the costs. If we invest in prevention and appraisal activities at an early stage, then failures and failure-costs reduce (which are high if prevention and appraisal are ignored), as shown in Fig. 13.3.

Figure 13.3 Benefits of investment on quality

Thus, by implementing quality evaluation programs, the following benefits are achieved:

- Customer is satisfied, as the end-product is of high quality.
- Productivity increases due to shorter SDLC cycle.
- Failures and failure costs reduce.
- Rework and cost of quality reduce.

13.5 Quality Control and Quality Assurance

It is clear that for a high-quality software, the removal of defects is necessary. This removal of defects takes place when quality control activities are adopted. Quality control involves a series of static and dynamic testing throughout the process, such that all the stages produce outputs according to the desired requirements.

The term 'quality control' was used in 1960s, basically for manufacturing products. The definition of quality control had a broad view at that time including the measurement of processes. In 1980s, the term 'quality assurance' became popular. Both these terms appear to be synonymous and confusing. But Pressman and Pankaj Jalote help to clarify the difference between the two.

According to Pankaj Jalote [3], *quality control focuses on finding and removing defects, whereas the main purpose of quality assurance is to verify that applicable procedures and standards are being followed.*

According to Pressman [7], *a key concept of quality control is that all work products have defined, measurable specifications to which we may compare the output of each process.*

Quality assurance consists of auditing and reporting functions of management. Based on the above definitions, we can define the two terms again in reference to software.

Quality control is basically related to software product such that there is minimum variation, according to the desired specifications. This variation is checked at each step of development. Quality control may include the following activities: reviews, testing using manual techniques or with automated tools (V&V).

Quality assurance is largely related to the process. In addition, quality assurance activities are in the management zone. Therefore, auditing and reporting of quality, based on quantitative measurements, are also performed. The goal is to inform the management about quality-related issues and supply relevant data. It is the management's responsibility to address these problems. Thus, software quality assurance (SQA) is defined as a planned and systematic approach to evaluate quality and maintain software product standards, processes, and procedures.

SQA includes the process of assuring that standards and procedures are established and are followed throughout the software acquisition life cycle. Compliance with agreed-upon standards and procedures is evaluated through process monitoring, product evaluation, and audits. Software development and control processes should include quality assurance approval points, where an SQA evaluation of the product may be done in relation to the applicable standards.

Quality assurance may include the following activities:

- Measuring the products and processes.
- Auditing the quality of every step in SDLC and the end-product.
- Verifying that the adopted procedures and standards are being followed.

13.6 Quality Management (QM)

We have discussed that quality assurance is a management activity performed by managers. At the managerial level, quality needs to be planned on a larger scale and all controlling and assurance activities are performed according to quality plans. The managerial root has given the term 'quality management' whose scope is larger than quality control and quality assurance. It is now an established way of managing the quality of a product.

The task of quality management is to

- plan suitable quality control and quality assurance activities.
- define procedures and standards which should be used during software development and verify that these are being followed by everyone.
- properly execute and control activities.

If quality at some point is not under control, then it is the responsibility of quality managers to manage the resources such that the required level of quality is achieved.

But there is more to quality management. The major role of QM is to develop a *quality culture* in the organization. Quality culture means that every member of the team is aware and conscious about the quality and is working towards a high-quality end-product. It is the responsibility of quality managers to encourage the team members to take responsibility for their work, not only to function correctly but also for improving the quality (in the sense of defects). Though QM specifies the quality specifications and standards, yet there are some aspects which cannot be standardized, e.g. elegance, readability, etc. are intangible aspects of software quality. Quality managers also encourage the team members to keep an eye on these intagible aspects of quality. Thus, every team member works in unison to achieve the common goal of high-quality software.

QM involves verifying and improving the software process. The quality of intermediate results at each stage and the final end-product is measured, reviewed, and analysed. If the target does not seem to be achievable with the current process, then problems and difficulties are reported to senior management, and finally the current process can be improved based on this feedback.

13.7 QM and Project Management

QM is different from project management (PM), in the sense that QM is not associated with a particular project, rather it is an organization-wide activity for many projects. On the other hand, PM is specific to a particular project and manages the cost schedule, resources, and quality of that project only.

13.8 Quality Factors

We discussed earlier that quality is not a single idea; rather it is a combination of many factors. But these factors are based on the project and the environment in which the software is to be developed. Discussed below are some of these factors.

Functionality The major visible factor for software quality is its functionality such that all the intended features are working.

Correctness Software should be correct in nature largely satisfying specifications and fulfilling user's objectives. Another measure of correctness may be the extent to which a software is fault-free.

Completeness A software product should be complete to the extent that all its parts are present and fully developed. This means that if the code calls a subroutine from an external library, the software package must provide reference to that library and all the required parameters must be passed. All the required input data must be available.

Efficiency The software should be efficient in its working, i.e. utilizing all resources.

Portability A software product should be operated on all well-known platforms and configurations.

Testability A software should be testable such that it is easy to debug and maintain.

Usability The software should be easy to use, operate, and understand in every manner that its user wants. The usability criterion may differ from project to project.

Reliability It is the ultimate factor that a user demands from any product. A software product should also be reliable in the sense that all its desired features must run forever without bugs; it should not stop after working for some years. This implies a time factor in which a reliable product is expected to perform correctly over a period of time. It also encompasses environmental considerations in which the product is required to perform correctly in whichever condition it finds itself—this is sometimes called *robustness*. The reliability factor is more in demand in case of real-time projects, where the system must work continuously without fail.

Reusability It is an important factor, as it can greatly reduce the effort required to engineer a software product. However, it is not easy to achieve.

Integrity It defines the extent to which an unauthorized access or a modification can be controlled in a computer system (either manually or with automated tools).

Maintainability The software should have the scope of maintaining it, if a bug appears or a new requirement needs to be incorporated.

13.9 Methods of Quality Management

We have discussed the meaning of quality management and the activities which are involved in it. There are largely two approaches to perform QM (see Fig. 13.4), which are discussed here.

13.9.1 Procedural Approach to QM

In this approach, quality is controlled by detecting defects by performing reviews, walkthroughs, inspections (static testing), and validation testing methods. For implementing this approach, the plan and procedures are defined for conducting reviews and testing. As discussed earlier that every testing plan is prepared at a certain point and according to the defined procedure in test documents, testing activities are executed again on defined points. For example, acceptance test plan is prepared at the time of requirement gathering and acceptance testing is performed at the time of validation testing. Moreover, verification is conducted after every step of SDLC.

Execution of all V&V activities are performed by software engineers with the software quality assurance (SQA) team, who identify the defects. In this way, quality is being controlled by identifying and removing defects in the software.

Figure 13.4 Methods of quality management

Software Quality Assurance Activities

SQA activities include *product evaluation* and *process monitoring*. Products are monitored for conformance to standards and processes are monitored for conformance to procedures. Audits are a key technique used to perform product evaluation and process monitoring.

Product evaluation assures that standards are being followed. SQA team assures that clear and achievable standards exist and then evaluates the software with the established standards. Product evaluation assures that the software product reflects the requirements of the applicable standard(s). Process monitoring ensures that appropriate steps to carry out the process are being

followed. SQA team monitors processes by comparing the actual steps carried out with those in the documented procedures.

The audit technique used by the SQA team looks at a process and/or a product in depth, comparing them with the established procedures and standards. Audits are used to review management, technical, and assurance processes to provide an indication of the quality and status of the software product. The purpose of an SQA audit is to assure that proper control procedures are being followed, that required documentation is maintained, and that the developer's status reports accurately reflect the status of an activity. After the audit, an audit report is prepared that consists of findings and recommendations to bring the development into conformance with the standards and/or procedures.

SQA Relationships to Other Assurance Activities

Some of the more important relationships of SQA to other management and assurance activities are described below.

Configuration management monitoring SQA assures that software configuration management (CM) activities are performed in accordance with the CM plans, standards, and procedures. It reviews the CM plans for compliance with the software CM policies and requirements, and provides follow-up for nonconformances. It also audits the CM functions for adherence to standards and procedures and prepares a report of its findings.

The CM activities monitored and audited by SQA include baseline control, configuration identification, configuration control, configuration status accounting, and configuration authentication. SQA also monitors and audits the software library. It assures that:

- Baselines are established and consistently maintained for use in subsequent baseline development and control.
- Software configuration identification is consistent and accurate with respect to the numbering or naming of computer programs, software modules, software units, and associated software documents.
- Configuration control is maintained such that the software configuration used in critical phases of testing, acceptance, and delivery is compatible with the associated documentation.
- Configuration status accounting is performed accurately, including the recording and reporting of data, reflecting the software's configuration identification, proposed changes to the configuration identification, and the implementation status of approved changes.

- Software configuration authentication is established by a series of configuration reviews and audits that exhibit the performance required by the SRS and the configuration of the software is accurately reflected in the software design documents.
- Software development libraries provide for proper handling of software code, documentation, media, and related data in their various forms and versions from the time of their initial approval or acceptance until they have been incorporated into the final media.
- Approved changes to baselined software are made properly and consistently in all products, and no unauthorized changes are ma

Verification and validation monitoring SQA assures verification and validation (V&V) activities by monitoring technical reviews, inspections, and walkthroughs. The SQA team's role in reviews, inspections, and walkthroughs is to observe, participate as needed, and verify that they were properly conducted and documented. SQA also ensures that any actions required are assigned, documented, scheduled, and updated.

Formal test monitoring SQA assures that formal software testings like acceptance testing, is done in accordance with plans and procedures. SQA reviews testing documentation for completeness and adherence to standards. The documentation review includes test plans, test specifications, test procedures, and test reports. SQA monitors testing and provides follow-up on non-conformances. By test monitoring, SQA assures software completeness and readiness for delivery.

The objectives of SQA in monitoring formal software testing are to assure that:

- The test procedures are testing the software requirements in accordance with the test plans.
- The test procedures are verifiable.
- The correct or 'advertised' version of the software is being tested (by SQA monitoring of the CM activity).
- Non-conformances occurring during testing (that is, any incident not expected in the test procedures) are noted and recorded.
- Test reports are accurate and complete.
- Regression testing is conducted to assure that all the non-conformances have been corrected.
- Resolution of all non-conformances takes place prior to the delivery of product.

Software Quality Assurance during SDLC

In addition to the general activities described above, there are phase-specific SQA activities that should be conducted during SDLC. The activities for each phase are described here.

Concept and initiation phase SQA should be involved in both writing and reviewing the management plan in order to assure that the processes, procedures, and standards identified in the plan are appropriate, clear, specific, and auditable.

Requirements phase During this phase, SQA assures that software requirements are complete, testable, and properly expressed as functional, performance, and interface requirements.

Architectural design phase SQA activities during the architectural (preliminary) design phase include:

- Assuring adherence to approved design standards as designated in the management plan.
- Assuring all software requirements are allocated to software components.
- Assuring that a testing verification matrix exists and is kept up-to-date.
- Assuring the interface control documents are in agreement with the standard, in form and content.
- Assuring the approved design is placed under configuration management.

Detailed design phase SQA activities during the detailed design phase include:

- Assuring the approved design standards are followed.
- Assuring the allocated modules are included in the detailed design.
- Assuring the results of design inspections are included in the design.

Implementation phase SQA activities during the implementation phase include the audit of:

- Status of all deliverable items.
- Configuration management activities.
- Non-conformance reporting and corrective action system.

Integration and test phase SQA activities during the integration and test phase include:

- Assuring readiness for testing of all deliverable items.
- Assuring that all tests are run according to test plans and procedures and that non-conformances are reported and resolved.
- Assuring that test reports are complete and correct.
- Certifying that testing is complete, and the software and its documentation are ready for delivery.

Acceptance and delivery phase As a minimum, SQA activities during the acceptance and delivery phase include assuring the performance of a final configuration audit to demonstrate that all deliverable items are ready for delivery.

13.9.2 Quantitative Approach to QM

In the procedural approach, V&V activities are performed to control the quality by detecting and removing the bugs, but quality at the end of a project cannot be assessed. It means, we have no confidence in quality management. Therefore, the quantitative approach that measures every activity in the life cycle, analyses the data and then, gives judgement about the level of quality. If there is a need to increase the level of quality, then auditing is done to check whether all the procedures and standards are being followed. Thus, this approach covers all the QA activities.

The overall effect of implementing software metrics for software quality can be increased if the whole process of collecting data and analysing things are put into a framework or systematic program of activities called the software metrics programme (SMP). We discuss a model for SMP given by Paul Goodman [34].

Paul Goodman Model for SMP

This model presents a framework for SMP in the lifecycle stages just like SDLC. The idea is to identify and implement the software metrics in the finer detailed intermediate stages. This model comprises of five stages as discussed here (Fig. 13.5).

Figure 13.5 Paul Goodman model for SMP

Initiation stage There are many triggers that can start a metrics program and there are many individuals within an organization who can recognize such a trigger. Some triggers are:

- Productivity/quality concerns
- Customer complaints
- Customer/management requirement for more information
- A cost-cutting environment

Some of the initiators are:

- IT management
- Senior management (non-IT)
- Quality or R&D function

Requirement definition The SMP cannot be directly implemented in an organization without any reason. Therefore, the first stage is to find and analyse the problems in an organization so that these problems can be removed or controlled. Thus, in this stage, we collect the problems and define the requirements where we need to stress. The requirements may be customer identification, market identification, late discovery of project, overflow of cost, etc.

Component design After analysing the requirements in the previous stage, we will have sub-projects or components which must be designed or worked

on for the organization. Every requirement is linked to business needs. For example, cost estimation, reliability prediction, etc. may be the components which must be worked on for the organization.

Implementation Having completed all the four stages, the last stage is to implement the program in an organization. So after preparing a program, it has to be sold to the customers. To help the customer, training sessions and procedures or written guidance addressing the application of measurement must be provided. Changes or tailoring the original program according to local conditions of the organization can also be done. Moreover, help should be extended to the customers on a day-to-day basis.

Software metrics programs can be implemented for a project as well as for a process, if required.

Major Issues for Quantitative Approach

The major issues for quantitatively managing quality for a project are:

Setting quality goal The quality goals can be set in a project in terms of defects. It may be set as the number of defects to be delivered in the last phase of validation testing, i.e. acceptance testing. One method to set the quality goals is to use the past data if the same process is used in similar kind of projects. Using the past data on similar projects, the number of defects in the current project can be estimated, as given by Jalote [3]:

$$\textit{Estimate for defects}(P) = \textit{defects}(SP) \times \textit{effort estimate}(P) \,/\, \textit{actual effort}(SP)$$

where P is the current project and SP is the similar project

Managing the software development process quantitatively The development process is managed towards achieving the quality goal by setting intermediate goals at every intermediate stage and working towards these intermediate goals.

13.10 Software Quality Metrics

Software quality metrics are a subset of software metrics that focus on the quality aspects of the product, process, and project. In general, software quality metrics are more closely associated with process and product metrics than with project metrics. Nonetheless, the project parameters such as the number of developers and their skill levels, the schedule, the size, and the organization structure certainly affect the quality of product. Software quality metrics can be divided further into end-product quality, and based on the findings, to engineer the improvements in both process and product quality. Moreover,

we should view quality from the entire software life cycle perspective and in this regard, we should include metrics that measure the quality level of the maintenance process as another category of software quality metrics. Thus, software quality metrics can be grouped into the following three categories in accordance with the software lifecycle: (i) product quality metrics, in-process quality metrics, and metrics for software maintenance.

Product Quality Metrics

In product quality, the following metrics are considered [42]:

Mean-time to failure (MTTF) MTTF metric is an estimate of the average or mean time until a product's first failure occurs. It is most often used with safety critical systems such as the airline traffic control system.

It is calculated by dividing the total operating time of the units tested by the total number of failures encountered.

Defect density metrics It measures the defects relative to the software size.

$$\textit{Defect density} = \textit{Number of defects / Size of product}$$

Thus, it is the defect rate metrics or the volume of defects generally used for commercial software systems. It is important for cost and resource estimate of the maintenance phase of the software lifecycle.

Customer problem metrics This metric measures the problems which customers face while using the product. The problems may be valid defects or usability problems, unclear documentation, etc. The problem metrics is usually expressed in terms of problems per user month (PUM), as given below:

$$PUM = \frac{\textit{Total problems reported by the customer for a time period}}{\textit{Total number of licensed months of the software during the period}}$$

where number of licensed months = number of installed licenses of the software × number of months in the calculation period.

PUM is usually calculated for each month after the software is released. Basically, this metric relates with the problem of software usage. The goal is to achieve a low PUM by:

(a) Improving the development process and reducing the product defects.

(b) Reducing the non-defect-oriented problems by improving all the aspects of a product.

(c) Increasing the number of installed licenses of the product.

Customer satisfaction metrics Customer satisfaction is usually measured through various methods of customer surveys via the five-point scale.

1. Very satisfied
2. Satisfied
3. Neutral
4. Dissatisfied
5. Very dissatisfied

Based on this scale, several metrics can be prepared like the percentage of completely satisfied customers or the percentage of satisfied customers.

In-process Quality Metrics

In-process quality metrics are less formally defined than the end-product metrics. Following are some metrics [42]:

Defect-density during testing Higher defect rates found during testing is an indicator that the software has experienced higher error injection during its development process. This metric is a good indicator of quality while the software is still being tested. It is especially useful to monitor subsequent releases of a product in the same development organization.

Defect-arrival pattern during testing The pattern of defect arrivals or the time between consecutive failures gives more information. Different patterns of defect arrivals indicate different quality levels in the field. The objective is to have defect arrivals that stabilize at a very low level or times between failures that are far apart.

Defect-removal efficiency

$$DRE = \frac{Defects\ removed\ during\ the\ month}{Number\ of\ \ problem\ arrivals\ during\ the\ month} \times 100$$

Metrics for Software Maintenance

In this phase, the goal is to fix the defects as soon as possible and with excellent quality. The following metrics are important in software maintenance:

Fix backlog and backlog management index Fix backlog metrics is the count of reported problems that remain open at the end of a month or a week. This metric can provide meaningful information for managing the maintenance process.

Backlog management index (BMI) is also the metric to manage the backlog of open unresolved problems.

$$BMI = \frac{Number\ of\ \ problems\ closed\ during\ the\ month}{Number\ of\ \ problem\ arrivals\ during\ the\ month} \times 100$$

If BMI is larger than 100, it means the backlog is reduced; if BMI is less than 100, the backlog is increased. Therefore, the goal is to have a BMI larger than 100.

Fix response time and fix responsiveness While fixing the problems, time-limit also matters according to the severity of the problems. There may be some critical situations in which the customers feel the risk due to defects in the software product, therefore high-severity defects require the response in terms of fixing them as early as possible. The fix response time metric is used for this purpose and is calculated as mean-time of high severity defects from open to closed. Thus, a short fix response time leads to customer satisfaction.

This metric can be enhanced from the customers' viewpoint. Sometimes, the customer sets a time-limit to fix the defects. This agreed-to fix time is known as *fix responsiveness metric*. The goal is to get high fix responsiveness to meet customer expectations and have highly satisfied customers.

Percent delinquent fixes For each fix, if the turn-around time greatly exceeds the required response time, then it is classified as delinquent.

$$\textit{Percent delinquent fixes} = \frac{\textit{Number of fixes that exceeded the response time criteria by severity level}}{\textit{Number of fixes delivered in a specified time}} \times 100$$

Fix quality It is the metric to measure the number of fixes that turn out to be defective. A fix is defective if it did not fix the reported problems or injected a new defect as a consequence of fixing. For real-time software, this metric is important for customer satisfaction. Fix quality is measured as a percent defective fixes. Percent defective fixes is the percentage of all fixes in a time interval that are defective.

13.11 SQA Models

Organizations must adopt a model for standardizing their SQA activities. Many SQA models have been developed in recent years. Some of them are discussed below.

13.11.1 ISO 9126

This is an international software quality standard from ISO. It aims to eliminate any misunderstanding between customers and developers. There may be a case that the customer is not able to communicate the requirements for the product to be developed. Moreover, the developer should also be able to understand the requirement and assess with confidence whether it is possible

to provide the product with the right level of software quality. Thus, there may be problems with both customers and developers which may affect the delivery and quality of the software project. ISO 9126 targets the common understanding between customers and developers.

ISO 9126 provides a set of six quality characteristics with their definitions and associated quality evaluation process to be used when specifying the requirements, and evaluating the quality of software products throughout their life cycle. The quality characteristics are shown in Fig. 13.6. The quality characteristics are accompanied by guidelines for their use. Each attribute is associated with one or more metrics, which allow a value for that attribute to be determined for a particular system.

Figure 13.6 ISO 9126 quality attributes

ISO 9126 defines the following guidelines for implementation of this model:

Quality requirements definition A quality requirement specification is prepared with implied needs, technical documentation of the project, and the ISO standard.

Evaluation preparation This stage involves the selection of appropriate metrics, a rating-level definition, and the definition of assessment criteria. Metrics are quantifiable measures mapped onto scales. The rating levels definition determines what ranges of values on these scales count as satisfactory or unsatisfactory. The assessment criteria definition takes as input a management criterion specific to the project and prepares a procedure for summarizing the results of the evaluation.

Evaluation procedure The selected metrics are applied to the software product and values are taken on the scales. Subsequently, for each measured value, the rating level is determined. After this, a set of rated levels are summarized, resulting in a summary of the software product quality. This summary is then

compared with other aspects such as time and cost, and the final managerial decision is taken, based on managerial criteria.

13.11.2 Capability Maturity Model (CMM)

It is a framework meant for software development processes. This model is aimed to measure the quality of a process by recognizing it on a maturity scale. In this way, matured organizations can be distinguished from immature ones. The maturity of a process directly affects the progress of a project and its results. Therefore, for a high-quality software product, the process adopted for development must be a matured one.

The development of CMM started in 1986 by the Software Engineering Institute (SEI) with MITRE Corporation. They developed a process maturity framework for improving software processes in organizations. After working for four years on this framework, the SEI developed the CMM for software. Later on, it was revised and is continuing to evolve.

CMM Structure

The structure of CMM consists of five maturity levels. These levels consist of key process areas (KPAs) which are organized by common features. These features in turn consist of key practices, decided in KPAs depending on these common features, as shown in Fig. 13.7. The maturity levels indicate the process capability. Key practices describe the activities to be done. Common features address implementations. In this way, by working on the activities described in key practices under some common feature, we achieve a goal under a KPA. In other words, each KPA identifies a cluster of related activities that, when performed collectively, achieve a set of goals considered important for enhancing the process capability.

Figure 13.7(a) CMM Structure

Maturity levels	Indicate	Process capability
Key process areas	Achieve	Goals
Common features	Address	Implementation/Institutionalization
Key practices	Describe	Infrastructure/Activities

Figure 13.7(b) CMM Structure

The structure of CMM is hierarchical in nature such that if one organization wishes to be on maturity level 3, then it must first achieve maturity level 2, and then work according to the goals of the next higher level. It means that we need to attain maturity in a sequenced order of all levels, as shown in Fig. 13.8. The process capability at each maturity level is shown in Table 13.1.

Figure 13.8 Maturity levels

Table 13.1 Process capability at each maturity level

Maturity level	Process Capability
1	—
2	Disciplined process
3	Standard consistent process
4	Predictable process
5	Continuously improving process

Maturity Levels

A brief description of the five maturity levels is given below.

Initial At this level, the process is chaotic or ad-hoc, where there is no control on development progress and success is purely dependent on individual efforts. There is no stable environment for development. The team members may not know the components or goals of development processes. The projects exceed the budget and schedule.

Repeatable Earlier project successes are used here and the lessons learned from past projects are incorporated in the current project. Basic project management processes are established so that cost, schedule, and other parameters can be tracked. However, organization-wide processes still do not exist.

Defined The management scope is widened to the organization here. Project management as well as process management starts. The standard processes are defined with documentations.

Managed A quantitative understanding of the processes is established in order to monitor and control them. Quantitative quality goals are also set for the processes. It means the processes are predictable and controlled.

Optimizing The process is continually improved through incremental and innovative technological changes or improvements. In this way, quantitative process improvement objectives for the organization is identified, evaluated, and deployed.

Key Process Areas

The KPAs and their corresponding goals at each maturity level are discussed in Tables 13.2, 13.3, 13.4, and 13.5.

Table 13.2 KPAs for level 2

KPAs, at this level, focus on the project's concerns related to establishing basic project management controls.	
KPA	***Goals***
Requirement Management	Document the requirements properly. Manage the requirement changes properly.
Software Project Planning	Ensure proper project plan including estimation and listing of activities to be done.
Software Project Tracking and Oversight	Evaluate the actual performance of the project against the plans during project execution. Action, if there is deviation from plan.
Software Sub-contract Management	Selects qualified software sub-contractors. Maintain ongoing communications between prime contractor and the software sub-contractor. Track the software sub-contractor's actual results and performance against its commitments.
Software Quality Assurance	Plan the SQA activities. Ensure that there are proper processes by conducting review and audits. Take proper actions, if projects fail.
Software Configuration Management	Identify work products and documents to be controlled in the project. Control the changes in the software.

Table 13.3 KPAs for level 3

KPAs, at this level, address both project and organizational issues, as the organization establishes an infrastructure that institutionalizes effective software engineering and management processes across all projects.	
KPA	***Goals***
Organization Process Focus	Coordinate process development and improvement activities across the organization. Compare software processes with a process standard. Plan organization-level process development and improvement activities.
Organization Process Definition	Standard software processes are defined and documented.
Training Program	Identify training needs of various team members and implementation of training programs.
Integrated Software Management	Tailor the project from the standard defined process. Manage the project according to defined process.
Software Product Engineering	Define, integrate, and perform software engineering tasks. Keep the work products consistent especially if there are changes.
Inter-group Coordination	Ensure coordination between different groups.
Peer Reviews	Plan peer review activities. Identify and remove defects in the software work products.

Table 13.4 KPAs for level 4

KPAs, at this level, focus on establishing a quantitative understanding of both the software process and the software work products being built.	
KPA	***Goals***
Quantitative Process Management	Plan the quantitative process management activities. Control the performance of the project's defined software process quantitatively. The process capability of the organization's standard software process is known in quantitative terms.
Software Quality Management	Set and plan quantitative quality goals for the project. Measure the actual performance of the project with quality goals and compare them with plans.

Table 13.5 KPAs for level 5

KPAs, at this level, cover the issues that both the organization and the projects must address to implement continuous and measurable software process improvement.	
KPA	***Goals***
Defect Prevention	Plan the defect prevention activities. Identify, prioritize, and eliminate the common causes of bugs.
Technology Change Management	Plan the technology changes to be incorporated in the project, if any. Evaluate the effect of new technologies on quality and productivity.
Process Change Management	Plan the process improvement activities such that an organization-wide participation is there. Measure the change of improvement in the process.

Common Features

As discussed above, all KPAs are organized by common features. The common features are attributes that indicate what to implement according to a KPA goal. The common features have been listed in Table 13.6.

Table 13.6 List of common features

Common Feature	What to Implement/Institutionalize
Commitment to Perform	Organization must take action to ensure that the process is established and will carry on.
Ability to Perform	Define the pre-conditions that must exist in the project or organization to implement the software process competently. Estimate resources, organizational structures, and training.
Activities Performed	Assign the team members and identify procedures to implement a key process area. Perform the work, track it, and take corrective actions as necessary.
Measurement and Analysis	Measure the process and analyse the results.
Verifying Implementation	Ensure that the activities are performed in compliance with the process that has been established. Implement reviews and audits by management and software quality assurance team.

Assessment of Process Maturity

The assessment is done by an assessment team consisting of some experienced team members. These members start the process of assessment by collecting information about the process being assessed. The information about the process can be collected in the form of maturity questionnaires, documentation, or interviews. For assessment, some representative projects of the organization are selected.

13.11.3 Software Total Quality Management (STQM)

Total quality management (TQM) is a term that was originally coined in 1985 by the Naval Air Systems Command to describe its Japanese-style management approach to quality improvement. TQM is a philosophical management strategy to have quality culture in the entire organization. TQM means that the organization's culture is defined by the constant attainment of satisfaction through an integrated system of tools, techniques, and training. Thus, TQM is defined as a quality-centered, customer-focused, fact-based, team-driven, senior-management-led process to achieve an organization's strategic imperative through continuous process improvement.

T = Total = *everyone* in the organization

Q = Quality = *customer satisfaction*

M = Management = *people* and *processes*

Total quality here means to provide *quality product* and *quality services* to the customer, i.e. quality in all aspects. TQM is based on the participation of all the members of an organization to improve processes, products, services, and the culture they work in. TQM is a customer-driven approach, wherein the quality is defined and judged by the customer. It focuses on continuous process improvement to achieve high quality of product (or service). Moreover, the emphasis is to achieve total quality throughout the entire business, not just in the product. Thus, quality is a part of every function in all the phases of a product's lifecycle.

The elements of a TQM system can be summarized as follows:

1. **Customer focus** The goal is to attain total customer satisfaction. It includes studying customers' needs, gathering customer requirements, and measuring and managing customer satisfaction.
2. **Process** The goal is to achieve continuous process improvement. Process includes both the business process and the product development process. Through process improvement, product quality will be enhanced.
3. **Human-side of quality** The goal is to create a company-wide quality culture. Focus areas include management commitment, total participation, employee empowerment, and other social, psychological, and human factors.
4. **Measurement and analysis** The goal is to drive continuous improvement in all quality parameters by the goal-oriented measurement system.

TQM has a few short-term advantages. Most of its benefits are long-term and come into effect only after running smoothly for some time. In large organizations, it may take several years before long-term benefits are realized.

Long-term benefits that may be expected from TQM are: higher productivity, increased morale, reduced costs, and greater customer commitment.

TQM originated in the manufacturing sector, but it has been successfully adopted by almost every type of organization. So it can also be applied to software development because we need to improve our software processes. Remember, we can't test quality in our software, we design in it. And the only way we can design quality in it is by continuously monitoring and improving our processes. As we studied in the previous sections, the software quality is maintained by the SQA team. However, SQA is not enough to achieve the quality standards demanded by the customer. It becomes important to adopt the TQM method to the entire software development organization. This is known as *software total quality management* (STQM). The STQM is also a fundamental cultural change that incorporates quality improvements in every aspect of the software organization. Therefore, SQA provides the methodology to assure quality, while STQM provides the framework to continually improve it. STQM fits well in the components of TQM discussed above.

Customer focus in software development As we know, requirement bugs constitute a large portion of the software bugs. Therefore, gathering customers' needs has become increasingly important in the software industry.

Process, technology, and development quality Given the customers' requirements, the central question is how to develop the software effectively so that it can meet the criterion of 'conformance to customers' requirements'. For this, a matured process and updated technologies are needed to achieve product quality.

Human-side of software quality In software development, at the operational level, TQM can be viewed as the integration of project, process, and quality management with the support of every development team member. During software development, managers must manage in-process quality in the way they manage schedule for quality improvement. Quality, process, and schedule management must be totally integrated for software development to be effective.

Measurement and analysis Software development must be controlled in order to have product delivery on time and within budget, which are the prime concerns for every software development. Software metrics help in measuring various factors during development and thereby, in monitoring and controlling them.

13.11.4 Six Sigma

Six sigma is a quality model originally developed for manufacturing processes. It was developed by Motorola. Six sigma derives its meaning from the field of statistics. Sigma is the standard deviation for a statistical population. Six sigma means, if one has six standard deviations between the mean of a process and the nearest specification limit, there will be practically no items that fail to meet the specifications. Therefore, the goal of this model is to have a process quality of that level. Eventually, six sigma evolved and was applied to other non-manufacturing processes. Today, you can apply six sigma to many fields like services, medical, and insurance procedures, call centers including software.

Six sigma is a way to achieve strategic business results emphasizing on lower costs with less number of defects. Six sigma processes will produce less than 3.4 defects per million opportunities. To achieve this target, it uses a methodology known as DMAIC with the following steps:

- **D**efine opportunities
- **M**easure performance
- **A**nalyse opportunity
- **I**mprove performance
- **C**ontrol performance

This methodology improves any existing business process by constantly reviewing and improving the process.

Six sigma is not just all about statistics. It can be applied to software problems which affects its quality. Six sigma can be applied to analyse the customer requirements and define business goals correspondingly. Its approach to customer requirements, if applied to software development projects, is fundamentally different than those typically practiced in software deployment efforts. It does not start by asking the customers about the requirements first. But it begins by analysing what we need to learn. There are six sigma tools which help in identifying the prioritization of functionalities to be delivered.

Summary

Quality is the concept from where the story of software testing begins. In simple words, we want quality in the software product, which is why we perform testing on it so that a high-quality product can be made. This chapter discusses the concept of quality, its benefits, its various factors, and the costs incurred in it.

In literature, three terms have been in use: quality control, quality assurance, and quality management; all these terms have been distinguished. Quality management is the term which

we use today in the broader sense. The two approaches for quality management, namely procedural approach and quantitative approach, have also been considered in detail. But for a successful high-quality product, we have to pass the project through both approaches. Some quality metrics to monitor and control the quality attributes have also been defined.

Let us review the important concepts described in this chapter:

- Quality is the degree to which a product or service possesses a desired combination of attributes.
- Software quality may be defined in the form of delivered defect density or defect rate, i.e. the number of defects per unit size (e.g. LOC, FP or other unit).
- Quality costs include: prevention costs, appraisal costs, and failure costs.
- Prevention costs are related with activities that identify the cause of defects and actions that are taken to prevent them.
- Appraisal costs include the costs of evaluating the quality of software products at some level, e.g. testing, in-process and inter-process inspection by implementing software metrics program (SMP), equipment calibrations, and maintenance.
- Failure costs include the costs to analyse and remove failures.
- Quality control focuses on finding and removing defects, whereas the main purpose of quality assurance is to verify that applicable procedures and standards are being followed.
- Software quality assurance (SQA) is defined as a planned and systematic approach to evaluate quality and maintain software product standards, processes, and procedures. SQA includes the process of assuring that standards and procedures are established and are followed throughout the software acquisition lifecycle.
- Quality management is an established way of managing the quality of a product wherein quality measures are planned on a larger scale and all controlling and assurance activities are performed according to quality plans.
- The overall effect of implementing software metrics for software quality can be increased if the whole process of collecting data and analysing them are put into a framework or systematic program of activities called software metrics program (SMP).
- Mean-time to failure (MTTF) metric is an estimate of the average or mean-time until a product's first failure occurs. It is calculated by dividing the total operating time of the units tested by the total number of failures encountered.
- Backlog management index (BMI) is the metric to manage the backlog of open and unresolved problems.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Quality is a _____ dimension concept.
   (a) single
   (b) two

(c) three
(d) multi

2. External failure costs are associated with failures that appear on the _____.
   (a) developer's site
   (b) customer's site
   (c) installation time
   (d) none of the above
3. Internal failure costs are those costs that appear on the _____prior to release.
   (a) developer's site
   (b) customer's site
   (c) installation time
   (d) none of the above
4. Quality control is basically related to software _____.
   (a) process
   (b) product
   (c) project
   (d) none of the above
5. Quality assurance is basically related to software _____.
   (a) process
   (b) product
   (c) project
   (d) none of the above
6. Scope of quality management is _____ than control or assurance.
   (a) smaller
   (b) same
   (c) larger
   (d) not related
7. Project management is related to _____ project.
   (a) one
   (b) two
   (c) three
   (d) many
8. Quality management is related to _____ project.
   (a) one
   (b) two
   (c) three
   (d) many

9. Defect density = _____ / size of product
   (a) Number of defects
   (b) Number of test cases
   (c) Number of attributes
   (d) none of the above
10. PUM should be _____.
   (a) high
   (b) low
   (c) average
   (d) none of the above
11. Customer satisfaction is usually measured through various methods of customer surveys via the _____ point scale.
   (a) 3
   (b) 4
   (c) 5
   (d) 6
12. BMI = (Number of problems _____ during the month / Number of problem arrivals during the month) × 100
   (a) closed
   (b) opened
   (c) none of the above
13. If BMI is larger than 100, it means the backlog is _____.
   (a) increased
   (b) reduced
   (c) remains same
   (d) none of the above

Review Questions

1. How does a process affect the quality of a product?
2. Take a project and prepare a list of all the categories of cost of quality in it.
3. What is the difference between quality management and project management? List the various activities under both.
4. How does failure costs reduce if we invest in prevention and appraisal activities? Explain with an example.
5. Differentiate quality control, quality assurance, and quality management.
6. Define the following metrics:
   (a) Defect density
   (b) PUM

(c) BMI
(d) Fix responsiveness
(e) Fix quality

7. Explain the various activities performed in the procedural approach for quality management.
8. Explain the various activities performed in the quantitative approach for quality management.
9. What is CMMI? What is the difference between CMMI and CMM?
10. Search the literature on SPICE quality model and compare it with CMM.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter

14

Testing Process Maturity Models

OBJECTIVES

After reading this chapter, you should be able to understand:

- Testing process should be measured for its maturity and improvement
- Various test maturity models for assessment and improvement of test process
- Test Improvement Model (TIM)
- Test Process Improvement (TPI) Model
- Test Organization Model (TOM)
- Test Maturity Model (TMM)

We have already discussed that software testing is more a systematic process than an intuitive one. Therefore, there should be ways or models for measuring its maturity. The world is working on measurements of the software development processes like CMM, but little attention has been given to measuring the maturity level of a testing process. Many organizations strive hard to get better results out of their testing process. But in the absence of a standardized testing process, they are not able to implement the best practices for software testing.

There are many questions regarding the implementation of a testing process. How should you organize and implement the various testing activities? How should these activities correlated with organizational goals? How should one measure the performance of one testing process? What are the key process areas to improve the current testing process? Whenever an organization plans to make investments in testing practices, the problem of test process assessment arises. We are dependent on conventional SQA models like CMM. However, CMM does not adequately address the testing process or testing-related issues. Thus, it is evident that while we have a standard for measuring the software development process, we also need to have a testing process maturity measurement model.

Some work has been done in the direction of developing the models of testing maturity. The prime purpose of these test maturity models was to provide a foundation for testing process improvement. Many models came up around 1995–96 for assessing the testing process maturity. Some of them have been developed based on the concepts already established in CMM. One of the

major models that has emerged is TMM (Testing Maturity Model), developed by Ilene Burnstein of Illinois Institute of Technology. Another model is TPI (Test Process Improvement), developed by Martin Pol and Koomen. These models provide a list of key process areas according to the test process activities and provide a formal basis for assessing any given system of software testing.

This chapter provides the details of all the major test process maturity models developed so far. The key process areas, levels, scope, and other details are discussed for these models.

14.1 Need for Test Process Maturity

The basic underlying goal for every process is to have continuous process improvement. The testing process should also be designed such that there is scope of its continuous improvement. It merges with our general goals of software development using software engineering, i.e. the software should be tested such that it is released on time, within budget, and is of high quality. But, test process improvement or test process maturity should be understood in the sense of testing terminology, also leading to the goals mentioned above.

There are some key process areas under which we should recognize the testing activities and concentrate on them so that these goals are achieved. However, there should be a mechanism or model which informs us whether the current practices of test process are sufficient or not. If they are not able to give the desired results, then we need to understand and work according to the key process areas and reach a desired level of that test process. The following are some of the reasons why test process maturity models are important:

- The progress and quality of the whole test process should be measured and the results should be used as input for further test process improvement.
- The testing process steps, i.e. test planning, design, execution, etc. are in place. There is a systematic flow of these activities in the organization.
- Testing maturity model provides important information to the testers and management, enabling them to improve the software quality.
- The model should be able to improve the process such that it provides information about the bug's source, i.e. the exact location of their existence, leading to minimization of correction costs.
- The model should be able to improve the process such that it detects high-priority or criticality bugs early.

14.2 Measurement and Improvement of a Test Process

To measure and improve an existing test process, the following steps are used [145]:

Determine target focus areas of test process Why do you want to measure or improve your current testing process? This question should be asked to realize what your long-term goal is. Without realizing the goals, you cannot improve the test process in reality. For example, you may have the target focus on shortening the time-frame of the test process. It may be possible that testing activities are not performed in order, so you want to have a proper test organization in place.

Analyse the testing practices in the current testing process Look for the strong and weak points of the current test process and analyse the aspects which are lacking.

Compare the current practices with a test maturity model Every test activity should be compared with a test maturity model in order to get the best test practices in a process.

Determine key process areas Based on the analysis done in the previous step, list the key process areas where the current process needs attention.

Develop a plan Depending on the key process areas recognized, develop a plan and suggest changes to nullify all the weak points of a process. Based on the suggestions, a modified test process is developed.

Implement the plan Implement the developed plan to improve the existing test process.

14.3 Test Process Maturity Models

Around 1996, many models were developed to measure the maturity of a test process. Some of the models are discussed here.

14.3.1 Testing Improvement Model

The testing improvement model (TIM) was developed by Ericson, Subotic, and Ursing [144]. TIM is used to identify the current state of practice in key areas, and it further suggests ways in which the weakness of a key area can be removed. TIM consists of a

(i) maturity model, and an (ii) Assessment procedure.

Maturity Model

TIM consists of four levels, each consisting of key areas. These are discussed below:

- *Baselining* A baseline work for all the testing activities is established. The test personnel are assigned responsibilities and every work is documented.
- *Cost-effectiveness* After establishing a baseline, efforts are made to improve the cost-effectiveness of testing activities. The focus is on early detection of bugs, training, and re-use.
- *Risk-lowering* Risk management is to have a plan that addresses how to deal with the undesired effects. For this purpose, it is required that the testing team communicates and cooperates with the development team. Ideally, the testing function should participate in all development work and meetings. At the risk-lowering level, metrics are put to use. Metrics are used for determining the fulfillment of test objectives, analysing the cost/benefits of test activities in order to justify expenses, and detecting fault-prone test objects.
- *Optimizing* Optimizing means to move towards continuous improvement. The test process should be able to measure the quality and have a base for improvement.

Key areas of TIM

The five key areas of TIM are:

- Organization
- Planning and tracking
- Testware
- Test cases
- Reviews

Each key area consists of related activities distributed over the four levels.

The Assessment Procedure of TIM

The assessment is performed as interviews, with strategically selected representatives from the organization. The results of the interviews are analysed and presented as 'maturity profile', where the organization's score for each key area is given. Based on the maturity profile, an improvement plan is developed, considering the needs and visions of the organization.

14.3.2 Test Organization Model (TOM)

This model emphasizes that the main hurdles in improving a test process is not only the technical issues, but organizational issues are also responsible. Therefore, while designing the test maturity model, organizational issues should also be considered which have been neglected in other models. Thus, the purpose of TOM is to identify and prioritize the organizational bottlenecks and generate solutions for these problems. This model consists of the following components [143]:

- Questionnaire to identify and prioritize symptoms
- Improvement suggestions

Questionnaire

It focuses on organization issues, as discussed above. Therefore, most of the questions will be answered by management personnel. There are 20 questions in total. This questionnaire serves to calculate a TOM level. The questions are answered with a score of 1–5. The questionnaire has four parts. Part 1 and Part 2 are meant for administrative and analysis purposes. Part 3 and Part 4 are used to calculate a TOM level. The various parts are discussed here:

Part 1 This part has questions on the organization, e.g. name of the organization, address, employee details, turnover, etc.

Part 2 It is the data about the assessor, so that the person in the organization can be contacted.

Part 3 The questions related to objectives and constraints are designed in this part. But the answers are in the form of a rating from 1–5. It aims to prioritize the objectives of the organization.

Part 4 Questions related to the symptoms of poor test process maturity are raised here. There are 20 symptoms. Again, the answers are in the form of a rating. Here, there are two ratings. First rating is the maturity score related to every symptom that shows various maturity levels of that symptom. The maturity scores are categorized in three parts: low (score 1), medium (score 3), and high (score 5). If you feel that your score is in between, then you can provide 2 or 4 as well. Second rating is the impact level of that symptom which is answered in the form of priority. Again, there are priority levels from 1 to 5. A priority 1 symptom can be ignored. A priority 5 is extremely important and must be addressed.

All ratings are added and a TOM level is calculated.

Improvement Suggestions

Based on the TOM level, the model provides testing improvements based on a library of 83 potential improvements. Based on the assessment data entered, the model may generate up to 70 prioritized improvement suggestions.

14.3.3 Test Process Improvement (TPI) Model

The TPI model was developed by Koomen and Pol in 1997 [140,141]. It contains the following parts (Fig. 14.1):

- Key process areas
- Maturity levels
- Test maturity matrix
- Checklist
- Improvement suggestions

Figure 14.1 TPI model structure

In the TPI model, all key areas are not considered equally important for the performance of the whole test process. We can map some dependencies between the different key areas and levels. This mapping has been represented as a *test maturity matrix* in this model. Moreover, the concept of checkpoints at every level has been introduced to verify that the activities have been implemented at a level. If a test process passes all the checkpoints of a certain level, then the process is classified at that level. *Improvement suggestions* have also been added to the model, giving instructions and suggestions for reaching a certain maturity level.

Key Process Areas

In each test process, certain areas need specific attention in order to achieve a well-defined process. These key areas are the basis for improving and structuring the test process. Within the TPI model, 20 key areas have been recognized to determine the maturity of a test process. These 20 areas are further categorized into four groups identified as cornerstones according to the Tmap methodology [142]. These cornerstones, within each test process, require some degree of attention. For a balanced test process, the development of the cornerstones should be in balance. The cornerstones are (see Fig. 14.2):

- Lifecycle (L) of test activities related to the development cycle.
- Good organization (O).
- Right infrastructure and tools (I).
- Usable techniques (T) for performing the activities.

Figure 14.2 TPI key area groups

Lifecycle-related key areas

Test strategy The test strategy goal is to detect critical bugs early in the lifecycle. Moreover, there should be a complete mapping defined in the strategy to show which requirements and quality risks are discovered by what tests.

Lifecycle model The selected lifecycle model must be clearly defined with phases. Activities must be defined under each phase and all the details of every activity like goals, inputs, processes, outputs, dependencies, etc. must be mentioned.

Moment of involvement It emphasizes the concept of early testing. Testing starts as soon as requirements specifications are ready, so as to have less number of bugs. Debugging also becomes easy and less costly.

Techniques-related key areas

Estimating and planning Estimate the resources needed for testing and plan all the test activities.

Test specification techniques Define a standardized test specification technique through which good quality test cases can be designed.

Static test techniques Define the techniques and plan for performing static testing.

Metrics Develop and define standard metrics to measure every test activity.

Infrastructure and tools-related key areas

Test automation/tools Testing tools can have significant benefits, like shorter lead-times and more testing flexibility. Recognize and deploy the tools required at various levels of testing.

Test environment It consists of the following components: procedures, hardware, software, means of communication, facilities for building and using databases and files. The environment has a significant influence on the quality, lead-time, and costs of the testing process.

Office environment It includes all the requirements related to the physical look of the environment where testing is being conducted.

Organization-related key areas

Commitment and motivation All the personnel issues which motivate the testers to be committed towards their responsibility are recognized.

Test functions and training Form a team of skilled persons and provide training, if required.

Scope of methodology The aim is to have a methodology which is sufficiently generic to be applicable in every situation, and contains enough details so that it is not necessary to rethink the same items again each time.

Communication Good communication channels facilitate smooth running of a test process and create good conditions to optimize the test strategy.

Reporting There should be a proper mechanism for reporting bug, status of quality, etc.

Defect management Track the lifecycle of a defect and analyse the quality trends in the detected defects. Such analysis is used, for example, to give well-founded quality advice, process improvement, etc.

Testware management Every activity of testing must be documented and managed in a standard template.

Test process management To control every activity/process, four steps are essential: planning, execution, maintaining, and adjusting. These steps need to be identified for every activity/process.

Evaluation Evaluate all the items being produced, such as SRS, SDD, etc. by testing the intermediate stages of lifecycle and after the product is ready.

Low-level testing Unit testing and integration testing are well-known low-level tests. Low-level testing is efficient because it requires little communication and often, the finder is both the error-producer as well as the one who corrects the defect.

Maturity Levels

There are four basic levels in TPI consisting of 14 scales. Each level consists of a specific range of scales. The maturity level is assigned to every KPA. Further, levels (A, B, C, D) are used to assign a degree of maturity to each key area.

The maturity levels with their allotted scales are:

- **Ad hoc (Scale 0)**
- **Controlled (Scale 1–5)**
  - (i) Defined process steps
  - (ii) Defined test strategy
  - (iii) Test specification techniques
  - (iv) Bug reporting
  - (v) Testware management
- **Efficient (Scale 6–10)** The testing process is automated and integrated in the development organization.
- **Optimizing (Scale 11–13)** Continuous improvement to optimize the organization is a part of the regular methods of the organization.

Test Maturity Matrix

If we want to look at the key areas which need attention and improvement, all the information on maturity of a key area can be maintained in a matrix consisting of key area and maturity levels with their scales, as shown in Fig. 14.3. The main purpose of the matrix is to show the strong and weak points of the current test process. This helps in prioritizing the actions to be taken for improvement.

Key Area	Ad hoc	Controlled					Efficient					Optimizing		
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Test Strategy														
Life Cycle Model														
...	Increasing Test Maturity →													

Figure 14.3 Test maturity matrix

Checkpoints

Checkpoints are the helping points to find what a maturity level demands. They are maintained in the form of a list of questions. If the answers to these questions are positive, it means the level is successfully achieved.

Improvement Suggestions

TPI model includes suggestions to improve the test process. These are different kinds of hints and ideas that will help to achieve a certain level of test maturity.

14.3.4 Test Maturity Model (TMM)

TMM was developed in 1996 at the Illinois Institute of Technology [2,3]. It has two major components:

1. A maturity model in which five maturity levels are distinguished (as in CMM)
2. An assessment model containing an assessment procedure, an assessment instrument/questionnaire, team training, and selection criteria

TMM Components

The TMM structure contains several components, which are described here. Figure 14.4 shows the framework of TMM.

Maturity levels Each maturity level represents a specific testing maturity in the evolution to a matured testing process. Each upper level assumes that the lower level has practices that have been executed.

There are five levels within TMM:

- Initial
- Phase definition
- Integration
- Management and measurement
- Optimization/defect prevention and quality control

Figure 14.4 Framework of TMM

Maturity goals To achieve a certain maturity level in the model, some goals have been identified that must be addressed. The maturity goals of TMM are the same as what we call the key process areas in CMM. Goals are identified for each level, except for level 1.

Maturity sub-goals For each maturity goal, the sub-goals that are more concrete are defined. They define the scope, boundaries, and accomplishments for a particular level.

Activities, tasks, and responsibilities (ATRs) To implement the sub-goals identified, there should be a defined set of activities and responsibilities. Therefore, a set of ATRs are designed for each sub-goal. The ATRs address implementation and organizational adaptation issues at a specific level, targeting to improve the testing capability. The responsibility for these activities and tasks is assigned to the three key participants in the testing process: managers, developers/testers, and users/clients.

Examples of ATRs are:

- Upper management provides adequate resources and funding from the committee for testing and debugging (manager view).

- Developers/testers work with managers to set testing and debugging goals (developers/testers view).
- The testing goals for each level of testing: unit, integration, system, and acceptance are set (developers/testers view).
- Users/clients meet with developers/testers to give their view of the nature of testing/debugging and policies. These goals and policies should represent their interests with respect to product quality (users/clients view).

TMM Levels

TMM has been deliberately designed similar to CMM. The idea is to ensure the growth in testing maturity is in tune with the growth in software capability maturity. The characteristics of five TMM levels are given below, as shown in Fig. 14.5.

Figure 14.5 Five levels of TMM

Initial level It is characterized by a random testing process. Testing activities are performed after coding in an ad-hoc way. The objective of testing at this level is to show only the functioning of the software.

Phase definition Testing and debugging are considered separate activities at this level. Testing still follows coding, but is a planned activity. The primary goal of testing at this maturity level is to show that the software meets its specifications.

Integration It assumes that testing is no longer a phase after coding; it is integrated into the entire software lifecycle and gets early testing consideration. Test objectives are established with respect to the requirements based on user and client needs and are used for test case plan, design, and success criteria. There is a test organization in place consisting of various persons with assigned responsibilities.

Management and measurement At this level, testing is recognized as a measured and quantified process. Testing starts as soon as the SRS is prepared, i.e. reviews

at all phases of the development process are now recognized as testing and quality control activities. Testware is maintained for reuse, defects are adequately logged.

Optimization, defect-prevention, and quality control At this level, the testing process is defined and managed. The test process can be measured for costs and other parameters and effectiveness can be monitored. The concept of bug-prevention instead of bug-detection and quality control are practiced. There are mechanisms put in place to fine-tune and improve the testing process.

Maturity goals and sub-goals As discussed above, TMM levels include various goals and under each goal, there are sub-goals. These goals and sub-goals are necessary in order to measure a test process maturity and correspondingly focus on those goals and sub-goals in which the process is deficient. If everything is defined in the model in the form of goals, sub-goals, and ATRs, then it becomes easy to measure the maturity of the process. The goals at each level are discussed below, as shown in Table 14.1.

Table 14.1 TMM maturity goals by level

Maturity Level	Goals
Level 1	--
Level 2	Develop and establish a test policy. Develop testing and debugging goals. Initiate a testing planning process. Institutionalize basic testing techniques and methods.
Level 3	Establish a software test organization. Establish a testing training program. Integrate testing into software life cycle. Control and monitor test process.
Level 4	Establish an organization-wide review program. Establish a test measurement program. Software quality evaluation.
Level 5	Application of process data for defect-prevention. Quality control. Test process optimization.

Level 2 – Phase Definition The maturity goals and sub-goals at Level 2 are given below.

Develop and establish a test policy An organization should clearly set the objectives and accordingly, develop a test policy. This goal aims to develop and establish a test policy and an overall test approach for the organization. The sub-goals, under this goal, are:

1. Define the organization's overall test objectives aligned with the overall business.
2. Objectives should be translated into a set of high-level key test performance indicators. The establishment of performance objectives and indicators provides clear direction and communication of expected and achieved levels of performance.
3. Define an overall test approach like the V-approach.
4. Identify the levels of testing in the test approach, such as unit, integration, and system testing levels and identify the goals, responsibilities, and main tasks at each level.

Develop testing and debugging goals This goal emphasizes that testing and debugging are two separate processes. Testing process now is a planned activity with identified objectives and test approach, rather than a random process. The sub-goals, under this goal, are given below:

1. Make separate teams for testing and debugging.
2. The teams must develop and record testing goals.
3. The teams must develop and record debugging goals.
4. The documented testing and debugging goals must be distributed to all project managers and developers.
5. Test plans should be based on these testing goals.

Initiate a test planning process If we want to make a process repeatable, defined, and managed, then it must be planned properly. The test planning may involve several activities, such as analysing risks, making the test strategy according to the test phase, and identifying risks and specifications for designing the test cases. The sub-goals, under this goal, are:

1. An organization-wide test planning team must be formed.
2. A framework for organization-wide test planning policies must be established and supported by the management.
3. A test plan template must be developed and made available to project managers and developers.
4. Test plan should be prepared with the inputs from the requirement specifications.

Institutionalize basic testing techniques and methods Some testing techniques and tools must be identified so that the test cases are designed and executed effectively. Examples of basic techniques and methods are black-box and white-box testing strategies, the use of a requirements traceability matrix, etc.

The sub-goals, under this goal, are:

1. An organization-wide test technology group must be formed to study, evaluate, and recommend a set of basic testing techniques and methods, and recommend a set of simple tools to support them.
2. Management must establish and encourage a set of policies that ensures recommended techniques and methods are consistently applied throughout the organization.

Test environment The idea under this goal is to establish and maintain an integrated software and hardware environment in which it is possible to execute the tests in a manageable and repeatable way. The major concern is that the environment should be closer to the actual one, so that more bugs can be found. The sub-goals, under this goal, are:

1. Test environments are specified and their availability on time is ensured in the projects.
2. For higher test levels, the test environment is as 'real-life' as possible.
3. Test environments are managed and controlled according to documented procedures.

Level 3 – Integration The maturity goals and sub-goals at Level 3 are given below.

Establish a software test organization A test organization including the various test persons with their assigned responsibilities is necessary, as testing includes many complex activities. The sub-goals under this goal are:

1. An organization-wide testing team must be established with a hierarchical order of various team members.
2. Roles and responsibilities must be defined for the testing team.
3. The testing team must participate in discussions with the customer and the developer to incorporate user needs, requirements, and other design issues into the testing process.

Establish a technical training program A technical training program for test planning, testing methods, standards, techniques, review process, and tools should be provided to the testing team. Training includes in-house courses, self-study, and mentoring programs. The sub-goals, under this goal, are:

1. Training needs, goals, and plans must be developed from time to time.
2. Management must establish an organizational training program, based on identified goals and plans.

3. An in-house training group must be established with the tools, facilities, and materials in place.

Integrate testing into the software lifecycle The early testing concept is recognized in this goal. All the team members of development and testing should realize at this level that the testing activity should be conducted parallel to all lifecycle phases in order to get a high-quality software. Testing activities must start as soon as the SRS is prepared. The sub-goals, under this goal, are:

1. Identify the phases of SDLC where testing will be performed in order to integrate the testing activities into the software lifecycle.
2. Develop a customized V-model, based on defined testing sub-phases.
3. Standard definitions must be developed for the testware to be used in the organization.
4. Testers must participate in review meetings like requirement specifications with analysts, design meetings with developers in order to get familiar with the customer, level requirements and all design issues. This will help testers in reviewing, planning, and designing every test activity.

Control and monitor the testing process Several controlling and monitoring activities ensure that the testing process proceeds according to the plan. Every test activity must be tracked and controlled according to the test plans. The sub-goals are:

1. The organization must develop mechanisms and policies to control and monitor the testing process.
2. A set of basic test process-related measurements must be defined, recorded, and distributed.
3. A set of corrective actions and contingency plans must be developed, recorded, and documented for use when the testing deviates significantly from what is planned.

Level 4 – Management and Measurement At this level, the meaning of 'testing activity' is expanded such that reviews, inspections, and walkthroughs at all phases of the lifecycle are also a part of it. Every testware is verified. This expanded definition of testing covers activities typically categorized as verification and validation activities. Moreover, the controlling and monitoring functions discussed in the previous level are now supported by an established test measurement program. Test cases and procedures are stored for reuse and regression testing. The maturity goals and sub-goals at Level 4 are discussed here.

Establish an organization-wide review program Reviews are conducted at all the phases of the lifecycle to identify, catalog, and remove defects from software work products and to test work products early and effectively. The sub-goals are:

1. The upper management must develop review policies, support the review process, and take responsibility for integrating them into the organizational culture.
2. The testing team and the SQA team must develop and document goals, plans, follow-up procedures, and recording mechanisms for reviews throughout the software lifecycle.
3. Items for review must be specified by the above-mentioned two teams.
4. Personnel must be trained so that they understand and follow proper review policies, practices, and procedures.

Establish a test measurement program A formalized test-measurement program should be prepared keeping in view the broad organizational goals. Measurements may include test progress, test costs, data on errors and defects, and product measures such as software reliability. The sub-goals are:

1. Organization-wide test measurement policies and goals must be defined.
2. A test measurement plan must be developed with mechanisms for data collection, analysis, and application.
3. Action plans that apply measurement results to test process improvements must be developed and documented.

Software quality evaluation The test-measurement program must be correlated with the quality goals of the software and these goals must be evaluated through the metrics developed. The sub-goals, under this goal, are:

1. Quality goals must be developed and known to all team members.
2. The testing process must be structured, measured, and evaluated to ensure that the quality goals can be achieved.

Level 5 – Optimization, Defect-prevention, and Quality control The maturity goals and sub-goals at Level 5 are given below:

Application of process data for defect-prevention At this level, organizations must be able to learn from their bug history and prevent the bugs before they occur. Bug-prevention is applied across all projects and across the organization. A defect-prevention team is responsible for the defect-prevention activities, interacting with developers to apply defect-prevention activities throughout the lifecycle. The sub-goals, under this goal, are given below:

1. A bug-prevention team must be established with management support.
2. Bugs history must be recorded at every phase of lifecycle.
3. A causal analysis mechanism must be established to identify the root causes of bugs.
4. Action plans must be developed through the interaction of managers, developers, and testers to prevent identified defects from recurring. These plans must be tracked.

Quality control At this level, organizations use statistical sampling, measurements of confidence levels, trustworthiness, and reliability goals to drive the testing process and consequently control the quality parameters. The sub-goals, under this goal, are:

1. SQA group must establish quality goals for software products such as product unit defectiveness, confidence levels, and trustworthiness.
2. Test managers must incorporate these quality goals into the test plans.
3. The test group must be trained in statistical methods.

Test process optimization There is scope of continuous improvement for any test process. Therefore, the test process is quantified and can be fine-tuned so that it is optimized. Optimizing the testing process involves evaluation of every test activity and adaptation of improvements. The sub-goals are:

1. Assign the responsibility to monitor the testing process and identify areas for improvement to a group.
2. A mechanism must be in place to evaluate new tools and technologies that may improve the capability and maturity of the testing process.
3. The effectiveness of the testing process must be continually evaluated in a measurable and optimal manner.

The key process areas (KPAs) of each level are discussed in the tables below.

Table 14.2 KPAs at Level 2

KPAs	Associated activities
Develop testing and debugging goals	Various goals, tasks, activities, and tools for each must be identified. Responsibilities for each must be assigned.
Initiate a testing planning process	Focus on initiating a planning process which involves stating objectives, analysing risks, outlining strategies, and developing test design specifications and test cases. Also addresses the allocation of recourses and responsibilities for testing on the unit, integration system, and acceptance levels.
Institutionalize basic testing techniques and methods	Emphasis on applying basic testing techniques and methods to improve test process capability across the organization.

Table 14.3 KPAs at Level 3

KPAs	Associated activities
Establish a software test organization	Establishment of a software test organization to identify a group of people who are responsible for testing.
Establish a testing training program	Establishment of a testing training program to ensure a skilled staff is available to the testing group.
Integrate testing into software life cycle	Integration of testing sub-phases into software life cycle. Also a mechanism must be established that allows testers to work with developers, which facilitates testing activity integration.
Control and monitor test process	Several controlling and monitoring activities are performed which provide visibility and ensure that the testing process proceeds according to plan.

Table 14.4 KPAs at Level 4

KPAs	Associated activities
Establish an organization-wide review program	Establishment of a review program to remove defects from software work products and to test work products early and effectively.
Establish a test measurement program	Organizational–wide test measurement policies and goals must be defined as well as action plans that apply measurement results to test process improvements must be developed and documented.
Software quality evaluation	Software quality evaluation involves defining measurable quality attributes and defining quality goals to evaluate software work products. Quality goals are tied to testing process adequacy.

Table 14.5 KPAs at Level 5

KPAs	Associated activities
Application of process data for defect-prevention	A defect-prevention team must be established with management support and the defects injected or removed must be identified and recorded during each phase.
Quality control	Organizations use statistical sampling, measurement of confidence levels, trustworthiness, and reliability goals to drive the testing process.
Test process optimization	Testing process is subjected to continuous improvement across projects and across the organization. A mechanism must be in place to evaluate new tools and technologies that may improve the capability and maturity of the testing process.

The Assessment Model

The TMM assessment model (TMM-AM) supports self-assessment of the testing process. The TMM-AM has three major components:

The tools for assessment It contains the tools for assessing the test process. For example, a questionnaire can be prepared. The questions designed may relate to maturity goals and process issues described at each level.

The assessment procedure The assessment procedure will give the assessment team guidelines on how to collect, organize, analyse, and interpret the data collected from questionnaires and personal interviews.

Training procedure A training procedure should be used to instruct the personnel in test process assessment.

SUMMARY

TMM models have been developed to measure and improve the maturity of a testing process. These models can be used along with SDLC maturity models like CMM. These models cannot replace SDLC maturity models, but they can assess an existing test process and suggest the areas where an organization can improve. Thus, an organization can use both SDLC and STLC maturity models in parallel.

This chapter discusses the importance, development, levels, and key process areas of various models: TPI, TOM, TIM, and TMM. The key process areas are important to recognize an organization, what its capabilities are, and what type of testing process it uses. Therefore, first identify the key process areas, check all of them with your organization, rate the organization at a maturity level, and strive for another higher level.

The major motive behind the maturity models is to produce good quality software by measuring the testing process. The test maturity models discussed in this chapter help in (i) improving the test process, (ii) attaining the next CMM level, (iii) identifying the areas to be improved, (iv) identifying the testing processes that can be adopted company-wide, and (v) providing a roadmap for implementing improvements.

Let us review the important concepts described in this chapter:

- The test maturity model informs whether the current practices of test process are sufficient or not.
- Testing maturity model provides important information to the testers and management, enabling them to improve the software quality.
- The testing improvement model (TIM) is used to identify the current state of practice in key areas and further, it suggests ways in which the weakness of a key area can be removed. It consists of a maturity model and an assessment procedure. The five key areas of TIM are: organization, planning and tracking, testware, test cases, and reviews.
- Test organization model (TOM) identifies and prioritizes the organizational bottlenecks and generates solutions for these problems. This model consists of the following components: (i) questionnaire to identify and prioritize symptoms and (ii) improvement suggestions.
- Test process improvement (TPI) model does not consider all key areas equally important for the performance of the whole test process. We can map some dependencies between the different key areas and levels. This mapping has been represented as a test maturity matrix in this model. Moreover, the concept of checkpoints at every level has

been introduced to verify that the activities have been completed at a level. It contains the following parts: key process areas, maturity levels, test maturity matrix, a checklist, and improvement suggestions.

- Within the TPI model, 20 key areas have been recognized to determine the maturity of the test process. These areas are further categorized into four groups identified as the cornerstones, namely: lifecycle (L) of test activities related to the development cycle, good organization (O), right infrastructure and tools (I), and usable techniques (T) for performing the activities.
- There are four basic maturity levels in TPI consisting of 14 scales. Each level consists of a specific range of scales. The maturity level is assigned to every KPA.
- In TPI, all the information of maturity of a key area can be maintained in a matrix consisting of key areas and maturity levels with their scales known as test maturity matrix.
- Test maturity model (TMM) has two major components: a maturity model with five maturity levels and an assessment model.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. The total number of questions in a TOM questionnaire are _____.
   (a) 30
   (b) 20
   (c) 10
   (d) 15
2. The TMM is characterized by _____ testing maturity levels.
   (a) 2
   (b) 3
   (c) 4
   (d) 5
3. Initial level is at TMM level _____.
   (a) 1
   (b) 2
   (c) 3
   (d) 4
4. Integration is at TMM level _____.
   (a) 1
   (b) 2
   (c) 3
   (d) 4

5. Management and measurement is at TMM level _____.
   (a) 1
   (b) 5
   (c) 3
   (d) 4
6. Optimization, defect-prevention, and quality control measurement is at TMM level _____.
   (a) 1
   (b) 5
   (c) 3
   (d) 4
7. Phase definition measurement is at TMM level _____.
   (a) 1
   (b) 2
   (c) 3
   (d) 4
8. The maturity goal 'develop testing and debugging goals' is at TMM level _____.
   (a) 1
   (b) 2
   (c) 3
   (d) 4
9. The maturity goal 'establish an organization-wide review program' is at TMM level _____.
   (a) 1
   (b) 2
   (c) 3
   (d) 4
10. The maturity goal 'control and monitor the testing process' is at TMM level _____.
   (a) 5
   (b) 2
   (c) 3
   (d) 4
11. The maturity goal 'quality control' is at TMM level _____.
   (a) 5
   (b) 2
   (c) 3
   (d) 4

Review Questions

1. What is the need of a test process maturity model?
2. Explain all the key areas mentioned in TIM.
3. Elaborate the assessment procedure in TIM.
4. What are the key areas and maturity levels mentioned in TPI?
5. What is the meaning of test maturity matrix in TPI? How do you develop this matrix?
6. How many levels are there in TMM?
7. Explain all the key process areas at each TMM level.
8. Compare and contrast the test process maturity models mentioned in the chapter.
9. What is the correlation between CMM and TMM?
10. How does TMM help in process improvement?
11. Develop a checklist for the activities to be done in all TMM levels, and find out if your organization follows them.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Part 5 Test Automation

CHAPTER

Chapter 15:
Automation and Testing Tools

The testing tools provide automation to almost every technique discussed in Part II. Therefore, testing tools can be adopted for all static and dynamic testing techniques to save time and money. This part discusses the need for automation and the various categories of testing tools. However, the testing tools are costly. Therefore, these tools should be selected carefully, keeping in mind their use in STLC. First, analyse the need for automation and then, go for the testing tool. One should not rely completely on testing tools. They cannot replace the whole testing activity. Thus, automated tools are merely a part of the solution and they are not a magical answer to the testing problem.

This part will make ground for the following concepts:

- Need for automation
- Static and dynamic tools
- Testing activity tools
- Costs incurred in testing tools
- Guidelines for automated testing
- Overview of some commercial tools

Chapter

15

Automation and Testing Tools

OBJECTIVES

After reading this chapter, you should be able to understand:

- Need for automating the testing activities
- Testing tools are not silver bullets to solve all the testing problems
- Static and dynamic testing tools
- Selection of testing tools
- Costs incurred in adopting a testing tool
- Guidelines of testing automation

As seen in Chapter 2, testing tools are a part of testing tactics. The testing techniques, either static or dynamic, can be automated with the help of software tools which can greatly enhance the software testing process. Testing today is not a manual operation but is assisted with many efficient tools that help the testers. Test automation is the most glamorous part of software testing. Testing automation is effective such that any kind of repetitive and tedious activities can be done by machines, and testers can utilize their time in more creative and critical work.

In the last decade, a lot of testing tools have been developed for use throughout the various SDLC phases. But the major part is the selection of tools from a pool of various categories of tools. Apart from the high cost of these tools, a single tool may not cover the whole testing automation. Thus, tools must be selected according to their application and needs of the organization. Automation is bound to fail if chosen for wrong reasons at the wrong places in SDLC.

However, automated testing should not be viewed as a replacement for manual testing. There is a misconception among professionals that software testing is easy as you only run the test cases with automated tools. This is the reason why newcomers prefer testing jobs. But the truth is that testers have many duties in the development and it is not only about running automated tools. There are many activities in the testing life cycle which cannot be automated and manual effort is required. Thus, automated tools are merely a part of the solution; they are not a magical answer to the testing problem. Automated testing tools will never replace the analytical skills required to conduct the test, nor will they replace manual testing. It must be seen as an enhancement to the manual testing process [12].

15.1 Need for Automation

If an organization needs to choose a testing tool, the following benefits of automation must be considered.

Reduction of testing effort In verification and validation strategies, numerous test case design methods have been studied. Test cases for a complete software may be hundreds of thousands or more in number. Executing all of them manually takes a lot of testing effort and time. Thus, execution of test suits through software tools greatly reduces the amount of time required.

Reduces the testers' involvement in executing tests Sometimes executing the test cases takes a long time. Automating this process of executing the test suit will relieve the testers to do some other work, thereby increasing the parallelism in testing efforts.

Facilitates regression testing As we know, regression testing is the most time-consuming process. If we automate the process of regression testing, then testing effort as well as the time taken will reduce as compared to manual testing.

Avoids human mistakes Manually executing the test cases may incorporate errors in the process or sometimes, we may be biased towards limited test cases while checking the software. Testing tools will not cause these problems which are introduced due to manual testing.

Reduces overall cost of the software As we have seen that if testing time increases, cost of the software also increases. But due to testing tools, time and therefore cost can be reduced to a greater level as testing tools ease the burden of the test case production and execution.

Simulated testing Load performance testing is an example of testing where the real-life situation needs to be simulated in the test environment. Sometimes, it may not be possible to create the load of a number of concurrent users or large amount of data in a project. Automated tools, on the other hand, can create millions of concurrent virtual users/data and effectively test the project in the test environment before releasing the product.

Internal testing Testing may require testing for memory leakage or checking the coverage of testing. Automation tools can help in these tasks quickly and accurately, whereas doing this manually would be cumbersome, inaccurate, and time-consuming.

Test enablers While development is not complete, some modules for testing are not ready. At that time, stubs or drivers are needed to prepare data, simulate environment, make calls, and then verify results. Automation reduces the effort required in this case and becomes essential.

Test case design Automated tools can be used to design test cases also. Through automation, better coverage can be guaranteed, than if done manually.

15.2 Categorization of Testing Tools

A single tool may not cover the whole testing process, therefore, a variety of testing tools are available according to different needs and users. The different categories of testing tools are discussed here:

15.2.1 Static and Dynamic Testing Tools

These tools are based on the type of execution of test cases, namely static and dynamic, as discussed in software testing techniques:

Static testing tools For static testing, there are static program analysers which scan the source program and detect possible faults and anomalies. These static tools parse the program text, recognize the various sentences, and detect the following:

- Statements are well-formed.
- Inferences about the control flow of the program.
- Compute the set of all possible values for program data.

Static tools perform the following types of static analysis:

Control flow analysis This analysis detects loops with multiple exits and entry points and unreachable code.

Data use analysis It detects all types of data faults.

Interface analysis It detects all interface faults. It also detects functions which are never declared and never called or function results that are never used.

Path analysis It identifies all possible paths through the program and unravels the program's control.

Dynamic testing tools These tools support the following:

- Dynamic testing activities.
- Many a times, systems are difficult to test because several operations are being performed concurrently. In such cases, it is difficult to anticipate conditions and generate representative test cases. Automated test tools enable the test team to capture the state of events during the execution of a program by preserving a snapshot of the conditions. These tools are sometimes called *program monitors*. The monitors perform the following [66] functions:

- List the number of times a component is called or line of code is executed. This information about the statement or path coverage of their test cases is used by testers.
- Report on whether a decision point has branched in all directions, thereby providing information about branch coverage.
- Report summary statistics providing a high-level view of the percentage of statements, paths, and branches that have been covered by the collective set of test cases run. This information is important when test objectives are stated in terms of coverage.

15.2.2 Testing Activity Tools

These tools are based on the testing activities or tasks in a particular phase of the SDLC. Testing activities can be categorized as [1]:

- Reviews and inspections
- Test planning
- Test design and development
- Test execution and evaluation

Now, we will discuss the testing tools based on these testing tasks.

Tools for review and inspections Since these tools are for static analysis on many items, some tools are designed to work with specifications but there are far too many tools available that work exclusively with code. In this category, the following types of tools are required:

Complexity analysis tools It is important for testers that complexity is analysed so that testing time and resources can be estimated. The complexity analysis tools analyse the areas of complexity and provide indication to testers.

Code comprehension These tools help in understanding dependencies, tracing program logic, viewing graphical representations of the program, and identifying the dead code. All these tasks enable the inspection team to analyse the code extensively.

Tools for test planning The types of tools required for test planning are:

1. Templates for test plan documentation
2. Test schedule and staffing estimates
3. Complexity analyser

Tools for test design and development Discussed below are the types of tools required for test design and development.

Test data generator It automates the generation of test data based on a user defined format. These tools can populate a database quickly based on a set of rules, whether data is needed for functional testing, data-driven load testing, or performance testing.

Test case generator It automates the procedure of generating the test cases. But it works with a requirement management tool which is meant to capture requirements information. Test case generator uses the information provided by the requirement management tool and creates the test cases. The test cases can also be generated with the information provided by the test engineer regarding the previous failures that have been discovered by him. This information is entered into this tool and it becomes the knowledge-based tool that uses the knowledge of historical figures to generate test cases.

Test execution and evaluation tools The types of tools required for test execution and evaluation are [1]:

Capture/playback tools These tools record events (including keystrokes, mouse activity, and display output) at the time of running the system and place the information into a script. The tool can then replay the script to test the system [22].

Coverage analysis tools These tools automate the process of thoroughly testing the software and provide a quantitative measure of the coverage of the system being tested. These tools are helpful in the following:

- Measuring structural coverage which enables the development and test teams to gain insight into the effectiveness of tests and test suites
- Quantifying the complexity of the design
- Help in specifying parts of the software which are not being covered
- Measure the number of integration tests required to qualify the design
- Help in producing integration tests
- Measuring the number of integration tests that have not been executed
- Measuring multiple levels of test coverage, including branch, condition, decision/condition, multiple conditions, and path coverage

Memory testing tools These tools verify that an application is properly using its memory resources. They check whether an application is:

- Not releasing memory allocated to it
- Overwriting/overreading array bounds
- Reading and using uninitialized memory

Test management tools Test management tools try to cover most of the activities in the testing life cycle. These tools may cover planning, analysis, and design. Some test management tools such as Rational's TestStudio are integrated with requirement and configuration management and defect tracking tools, in order to simplify the entire testing life cycle [12].

Network-testing tools There are various applications running in the client-server environments. However, these applications pose new complexity to the testing effort and increases potential for errors due to inter-platform connectivity. Therefore, these tools monitor, measure, test, and diagnose performance across an entire network including the following:

- Cover the performance of the server and the network
- Overall system performance
- Functionality across server, client, and the network

Performance testing tools There are various systems for which performance testing is a must but this becomes a tedious job in real-time systems. Performance testing tools help in measuring the response time and load capabilities of a system.

15.3 Selection of Testing Tools

The big question is how to select a testing tool. It may depend on several factors. What are the needs of the organization; what is the project environment; what is the current testing methodology; all these factors should be considered when choosing testing tools. Some guidelines to be followed by selecting a testing tool are given below.

Match the tool to its appropriate use Before selecting the tool, it is necessary to know its use. A tool may not be a general one or may not cover many features. Rather, most of the tools are meant for specific tasks. Therefore, the tester needs to be familiar with both the tool and its uses in order to make a proper selection.

Select the tool to its appropriate SDLC phase Since the methods of testing changes according to the SDLC phase, the testing tools also change. Therefore, it is necessary to choose the tool according to the SDLC phase, in which testing is to be done.

Select the tool to the skill of the tester The individual performing the test must select a tool that conforms to his skill level. For example, it would be inappropriate for a user to select a tool that requires programming skills when the user does not possess those skills.

Select a tool which is affordable Tools are always costly and increase the cost of the project. Therefore, choose the tool which is within the budget of the project. Increasing the budget of the project for a costlier tool is not desired. If the tool is under utilization, then added cost will have no benefits to the project. Thus, once you are sure that a particular tool will really help the project, then only go for it otherwise it can be managed without a tool also.

Determine how many tools are required for testing the system A single tool generally cannot satisfy all test requirements. It may be possible that many test tools are required for the entire project. Therefore, assess the tool as per the test requirements and determine the number and type of tools required.

Select the tool after examining the schedule of testing First, get an idea of the entire schedule of testing activities and then decide whether there is enough time for learning the testing tool and then performing automation with that tool. If there is not enough time to provide training on the tool, then there is no use of automation.

15.4 Costs incurred in Testing Tools

Automation is not free. Obviously employing the testing tools incur a high cost. Moreover, before acquiring the tools, significant work is required. Following are some facts pertaining to the cost incurred in testing tools [22, 12]:

Automated script development Automated test tools do not create test scripts. Therefore, a significant time is needed to program the tests. Scripts are themselves programming languages. Thus, automating test execution requires programming exercises.

Training is required It is not necessary that the tester will be aware of all the tools and can use them directly. He may require training regarding the tool, otherwise it ends up on the shelf or implemented inefficiently. Therefore, it becomes necessary that in a new project, cost of training on the tools should also be included in the project budget and schedule.

Configuration management Configuration management is necessary to track large number of files and test related artifacts.

Learning curve for the tools There is a learning curve in using any new tool. For example, test scripts generated by the tool during recording must be modified manually, requiring tool-scripting knowledge in order to make the script robust, reusable, and maintainable.

Testing tools can be intrusive It may be necessary that for automation some tools require that a special code is inserted in the system to work correctly and to be integrated with the testing tools. These tools are known as *intrusive tools* which require addition of a piece of code in the existing software system. Intrusive tools pose the risk that defects introduced by the code inserted specifically to facilitate testing could interfere with the normal functioning of the system.

Multiple tools are required As discussed earlier, it may be possible that your requirement is not satisfied with just one tool for automation. In such a case, you have to go for many tools which incur a lot of cost.

15.5 Guidelines for Automated Testing

Automation is not a magical answer to the testing problem. Testing tools can never replace the analytical skills required to conduct testing and manual testing. It incurs some cost as seen above and it may not provide the desired solution if you are not careful. Therefore, it is necessary that you carefully plan the automation before adopting it. Decide which tool and how many tools are required, how much resources are required including the cost of the tool and the time spent on training.

Discussed below are the guidelines to be followed if you have planned for automation in testing.

Consider building a tool instead of buying one, if possible It may not be possible every time. But if the requirement is small and sufficient resources allow, then go for building the tool instead of buying, after weighing the pros and cons. Whether to buy or build a tool requires management commitment, including budget and resource approvals.

Test the tool on an application prototype While purchasing the tool, it is important to verify that it works properly with the system being developed. However, it is not possible as the system being developed is often not available. Therefore, it is suggested that if possible, the development team can build a system prototype for evaluating the testing tool.

Not all the tests should be automated Automated testing is an enhancement of manual testing, but it cannot be expected that all test on a project can be automated. It is important to decide which parts need automation before going for tools. Some tests are impossible to automate, such as verifying a printout. It has to be done manually.

Select the tools according to organizational needs Do not buy the tools just for their popularity or to compete with other organizations. Focus on the needs of the organization and know the resources (budget, schedule) before choosing the automation tool.

Use proven test-script development techniques Automation can be effective if proven techniques are used to produce efficient, maintainable, and reusable test scripts. The following are some hints:

1. Read the data values from either spreadsheets or tool-provided data pools, rather than being hard-coded into the test-case script because this prevent test cases from being reused. Hard-coded values should be replaced with variables and whenever possible read data from external sources.
2. Use modular script development. It increases maintainability and readability of the source code.
3. Build library of reusable functions by separating the common actions into shared script library usable by all test engineers.
4. All test scripts should be stored in a version control tool.

Automate the regression tests whenever feasible Regression testing consumes a lot of time. If tools are used for this testing, the testing time can be reduced to a greater extent. Therefore, whenever possible, automate the regression test cases.

15.6 Overview of Some Commercial Testing Tools

Mercury Interactive's WinRunner It is a tool used for performing functional /regression testing. It automatically creates the test scripts by recording the user interactions on GUI of the software. These scripts can be run repeatedly whenever needed without any manual intervention. The test scripts can also be modified if required because there is support of Test Script language (TSL) with a 'C' like syntax. There is also provision for bringing the application to a known state if any problem has occurred during automated testing. WinRunner executes the statements by default with an interleaving of one second. But if some activities take more time to complete, then it synchronizes the next test cases automatically by waiting for the current operations to be completed.

Segue Software's SilkTest This tool is also for functional/regression testing. It supports 4Test as a scripting language which is an object-oriented scripting language. SilkTest has a provision for customized in-built recovery system

which helps in continuing the automated testing even if there is some failure in between.

IBM Rational SQA Robot It is another powerful tool for functional/regression testing. Synchronization of test cases with a default delay of 20 seconds is also available.

Mercury Interactive's LoadRunner This tool is used for performance and load testing of a system. Generally, the tool is helpful for client/server applications of various parameters with their actual load like response time, the number of users, etc. The major benefit of using this tool is that it creates virtual users on a single machine and tests the system on various parameters. Thus, performance and load testing is done with minimum infrastructure.

Apache's JMeter This is an open-source software tool used for performance and load testing.

Mercury Interactive's TestDirector TestDirector is a test management tool. It is a web-based tool with the advantage of managing the testing if two teams are at different locations. It manages the test process with four phases: specifying requirements, planning tests, running tests, and tracking defects. Therefore, there is advantage that the tests are planned as per the requirements evolved. Once the test plan is ready, the test cases are executed. Defect-tracking can be done in any phase of test process. During defect-tracking, new bug can be reported; assign responsibility to someone for bug repair; assign priority to the bug; bug repair status can be analysed, etc. This tool can also be integrated with LoadRunner or WinRunner.

SUMMARY

Testing tools are a big aid to the testers. Testing activities, if performed manually, may take a lot of time; thus there is need for automating the testing activities. This chapter discusses the importance and need of testing automation and various types of testing tools. The tools are mainly categorized as static and dynamic testing tools.

Testing tools are helpful to the testers, but they cannot replace all the manual testing activities. There are some places where we have to perform manual testing.

There was a time when it was thought that testing tools will fasten the speed of software development and the manual effort will be zero. But there is no silver bullet like this. The guidelines for automated testing are also provided at the end.

Let us review the important concepts described in this chapter:

- Testing tools are based on the type of execution of test cases, namely static and dynamic.

- Static tools perform control flow analysis, data use analysis, interface analysis, and path analysis.
- Automated test tools enable the test team to capture the state of events during the execution of a program by preserving a snapshot of conditions. These tools are sometimes called program monitors.
- Testing activity tools are based on the testing activities or tasks in a particular phase of the SDLC.
- The complexity analysis tools analyse the areas of complexity and provide indication to testers.
- Code comprehension tools help inspection team to analyse the code extensively by understanding dependencies, tracing program logic, viewing graphical representations of the program, and identifying the dead code.
- Test data generator automates the generation of test data based on a user-defined format.
- Test case generator uses the information provided by the requirement management tool and creates the test cases.
- Capture/playback tools record events at the time of running the system and place the information into a script which can be run by the tester later.
- Test coverage analysis tools provide a quantitative measure of the coverage of the system being tested.
- Memory testing tools verify that an application is properly using its memory resources.
- Test management tools try to cover most of the activities in the testing life cycle. These tools may cover planning, analysis, and design.
- Network testing tools monitor, measure, test, and diagnose performance across an entire network.
- Performance testing tools help in measuring the response time and load capabilities of a system.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Tools that enable the test team to capture the state of events during the execution of a program by preserving a snapshot of conditions are called _____.
   (a) monitor program
   (b) event program
   (c) snapshot program
   (d) code comprehension

2. Tools that help in understanding dependencies, tracing program logic, viewing graphical representations of the program, and identifying the dead code are called _____.
   (a) monitor program
   (b) event program
   (c) code comprehension
   (d) none of the above
3. Tools that automate the generation of test data based on a user-defined format are called _____.
   (a) test data generator
   (b) test case generator
   (c) code comprehension
   (d) none of the above
4. The tools that record events at the time of running the system and place the information into a script are called _____.
   (a) test data generator
   (b) test case generator
   (c) coverage analysis tools
   (d) capture playback tools
5. Tools that automate the process of thoroughly testing the software and provide a quantitative measure of the coverage of the system being tested are called _____.
   (a) test data generator
   (b) test case generator
   (c) coverage analysis tools
   (d) capture playback tools
6. WinRunner is a _____ tool.
   (a) test management
   (b) functional/regression testing
   (c) performance testing
   (d) none of the above
7. JMeter is a _____ tool.
   (a) test management
   (b) functional/regression testing
   (c) performance testing
   (d) none of the above
8. TestDirector is a _____ tool.
   (a) test management

(b) functional/regression testing
(c) performance testing
(d) none of the above

9. _____ is not a functional/regression testing tool.
   (a) JMeter
   (b) TestDirector
   (c) WinRunner
   (d) LoadRunner

Review Questions

1. What is the need of automating the testing activities?
2. What is the difference between static and dynamic tools?
3. Analyse and report which tools are available for test planning?
4. Analyse and report which tools are available for test design?
5. What are the guidelines for selecting a testing tool?
6. What are the costs incurred in adopting a testing tool?
7. Is a single testing tool sufficient for all testing activities?
8. Make a list of some important testing tools which are open-source on the Internet. Use them for testing your software.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Part 6

Testing for Specialized Environments

CHAPTERS

Chapter 16:
Testing Object-Oriented Software

Chapter 17:
Testing Web-based Systems

Testing in specialized environments needs more attention with more specialized techniques. This part is devoted to software testing in various specialized environments. Two specialized environments have been considered: testing for object-oriented software and testing for Web-based systems.

The major challenge of testing in specialized environments is that the meaning of different techniques changes according to the environment. For example, unit testing in general loses its meaning in object-oriented testing. Similarly, in Web-based testing, we first need to decide what a unit will be. Thus, using basic testing techniques, we need to explore the issues in current environment and apply these techniques correspondingly.

Some issues arise due to the environment in which the software is currently being developed and tested. For example, in object-oriented software, inheritance testing is a major issue. Therefore, techniques must be devised for inheritance testing. Similarly, in Web-based software, security of application is a major requirement and therefore, security testing becomes mandatory. Thus, the methods for security testing must be known for this testing. In this way, many issues need to be considered while testing specialized-environment software.

This part will make ground for the following concepts:

- Differences between traditional software and specialized-environment software (object-oriented software and Web-based software)
- Challenges in testing object-oriented software and web-based software
- Verification and validation of object-oriented software and Web-based software
- Special issues to be tested due to the specialized environment

Chapter

16

Testing Object-Oriented Software

OBJECTIVES

After reading this chapter, you should be able to understand:

- Issues in object-oriented testing
- Strategy for testing object-oriented software: Verification of UML model and validation
- Issues in testing a class
- Feature-based testing of class
- State-based testing of class
- Issues in testing inheritance
- Incremental testing method for inheritance
- Integration testing of object-oriented software
- Thread-based integration testing
- Implicit control flow-based integration testing
- UML-based testing and system testing based on use-cases

Object-oriented technology has evolved rapidly in the last two decades. Today, everyone is rushing out to adopt this technology. Object-oriented paradigm has affected almost every organization. The object model based on data abstraction, rather than function abstraction, manages the complexity, inherent in many different kinds of systems. This data-abstraction based on objects helps us to model the problem domain in a better understandable format. Thus, object-oriented technology is considered a natural way to understand and represent the problems. Further, this technology has given birth to the concept of reusability in software design. The concept of reusability leads to faster software development and high-quality programs.

Object-oriented technology (OOT) was established with a number of object-oriented programming languages. Today, OO paradigm is embedded in the complete software engineering approach. Software built with OO follows all the steps of *OO software engineering* (OOSE). Modeling, analysis, design, and testing of OO software is different as compared to the structured approach.

Testing an OO software is more challenging. Most of the testing concepts lose their meaning in OO technology, e.g. a unit in OOT is not a module but a class, and a class cannot be tested with input-output behaviour as in unit testing of a module. Thus, both testing strategies and techniques change in case of OO software.

16.1 OOT Basics

The major reason for adopting OOT and discarding the structured approach is the complexity of the software. Structured approach is unable to reduce the complexity as the software size increases, thereby increasing the schedule and cost of the software. On the other hand, OOT is able to tackle the complexity and models the software in a natural way.

In the structured approach, the emphasis is on action, not on data. But data is important for program existence. Global data is a structured approach wherein data can be corrupted by anyone. In OOT, data is hidden, not to be accessed by everyone. Functions in structured approach are not able to model the real world very well. But when we model the same problem with OOT, the modeling is quite obvious and natural. Let us review some of the basic concepts and terminology before discussing the testing issues of OO software.

16.1.1 Terminology

Object Grady Booch defined objects as [36]: 'Objects are abstraction of entities in the real world representing a particularly dense and cohesive clustering of information. It can be defined as a tangible entity that exhibits some well-defined behaviour. Objects serve to unify the ideas of algorithmic and data abstraction'.

Thus, objects are individual entities formed from two components: state and behaviour. The state is modeled by a data structure. The behaviour is modeled by a set of operations (or actions) which manipulate the data structure to obtain the required change of state and result. Together, they form a single coherent unit. The behaviour is invoked by sending a *message* to the object requesting a particular service.

Class All objects with the same structure and behaviour can be defined with a common blueprint. This blueprint called a class, defines the data and behaviour common to all objects of a certain type. Thus class is a template that describes collection of similar objects, e.g. vehicle is a class and car is one object of class vehicle. The operations provided by a class are also known as *features*, and each item of data defined by a class is a *data-member*. Any class that invokes the features of another class is known as a *client* of the class providing the features.

Encapsulation A class contains data as well as operations that operate on data. In this way, data and operations are said to be encapsulated into a single entity. Encapsulation avoids the accidental change in data such that no one can directly access the object's data.

Abstraction Shaw [127] defines abstraction as: 'A simplified description, or specification of a system that emphasizes some of the system's details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the user and suppresses details that are immaterial to him'.

Inheritance It is a strong feature of OOT. It provides the ability to derive new class from existing classes. The derived class inherits all the data and behaviour (operations) of the original base class. Moreover, derived class can also add new features, e.g. employee is the base class and we can derive manager as derived class.

Polymorphism This feature is required to extend an existing OO system. Polymorphism means having many forms and implementations of a particular functionality. For example, when a user wants to draw some shape (line, circle, etc.), the system issues a draw command. The system is composed of many shapes, each of them having the behaviour to draw itself. So when the user wants to draw a circle, the circle object's draw command is invoked. By using polymorphism, the system figures out, as it is running, which type of shape is being drawn [37].

16.1.2 Object-Oriented Modeling and UML

Modeling is the method to visualize, specify different parts of a system, and the relationship between them. Modeling basically divides the complex problem into smaller understandable elements and provides a solution to the problem. In object-oriented development, *Unified modeling language* (UML) has become the *de facto* standard for analysis and design of OOS. UML provides a number of graphical tools that can be used to visualize a system from different viewpoints. The views generally defined in it are:

User view This view is from the user's viewpoint which represents the goals and objectives of the system as perceived by the user. It includes use-cases and use-case diagram as a tool for specifying the objectives.

Structural view This view represents the static view of the system representing the elements that are either conceptual or physical. Class diagrams and object diagrams are used to represent this view.

Behavioural view This view represents the dynamic part of the system representing behaviour over time and space. The tools used for behavioral view are: collaboration diagrams, sequence diagrams, state chart diagrams, and activity diagrams.

Implementation view This view represents the distribution of logical elements of the system. It uses component diagrams as a tool.

Environmental view This view represents the distribution of physical elements of the system. It depicts nodes which are part of the physical hardware for deployment of the system. It uses deployment diagram as a tool.

Let us review all the graphical tools used in the UML modeling. It is necessary to understand them as you need to verify and validate the OO software development through these models.

Use-case model Requirements for a system can be elicited and analysed with the help of a set of scenarios that identify how to use the system. The following components are used for this model:

Actors An actor is an external entity or agent that provides or receives information from the system. Formally, an actor is anything that communicates with the system or product and that is external to the system itself. An actor is represented as a stick diagram.

Use-case A use-case is a scenario that identifies and describes how the system will be used in a given situation. Each use-case describes an unambiguous scenario of interactions between actors and the system. Thus, use-case define functional and operational requirements of the system. A use-case is represented as

A complete set of use-cases describe all the means to use a system. Use-cases can be represented as use-case templates and use-case diagrams.

Use-case templates They are prepared as structured narrative mentioning the purpose of use-case, actors involved, and the flow of events.

Use-case diagrams These show the visual interaction between use-cases and actors.

Class diagrams A class diagram represents the static structure of the system using classes identified in the system with associations between them. In other words, it shows the interactions between the classes in the system.

Object diagrams An object diagram represents the static structure of the system at a particular instance of time. It shows a set of objects and their relationships at a point of time.

Class-responsibility-collaboration (CRC) diagrams This diagram represents a set of classes and the interaction between them using associations and messages sent and received by the classes. These diagrams are used to understand the messages exchanged between the classes.

CRC model index card This card represents a class. The card is divided into three sections:

- On the top of the card, name of class is mentioned.
- In the body of the card on the left side, responsibilities of the class are mentioned.
- In the body of the card on the right side, the collaborators (relationship between classes) are mentioned.

Sequence diagram It is a pictorial representation of all possible sequences along the time-line axis to show the functionality of one use-case, e.g. for the use-case *Deposit the money in a bank*, there are sequences of events–customer goes to cashier with cash and form, submits it to the cashier, cashier receives the cash and form, deposits the amount and updates the balance, returns receipt to the customer.

State chart diagrams These diagrams represent the states of reactive objects (which respond to external events) and transitions between the states. A state chart depicts the object's initial state where it performs some activity and transitions after which the object enters another state. Thus, these diagrams represent the behaviour of objects with respect to their states and transitions.

Activity diagrams It is a special case of a state chart diagram, wherein states are activity states and transitions are triggered by the completion of activities.

Component diagrams At the physical level, the system can be represented as components. The components can be in the form of packages or sub-systems containing classes. The component diagrams show the set of components and their relationships.

Deployment diagrams These diagrams are also at the physical level. After the components and their interaction have been identified in the component diagram, the components are deployed for execution in the actual hardware nodes identified according the implementation. Thus, these diagrams depict the mappings of components to the nodes involved in the physical implementation of a system.

16.2 Object-Oriented Testing

It is clear that object-oriented software is different in many ways as compared to the conventional procedural software. OO software is easy to design but difficult to test. As dimensions of OOS increase in comparison to procedural software, the testing of OOS should also be broadened such that all the properties of OOT are tested.

16.2.1 Conventional Testing and OOT

Firesmith [67] has examined the differences between conventional testing and object-oriented software testing in some depth. The difference between unit testing for conventional software and object-oriented software arises from the nature of classes and objects. We know that unit testing only addresses the testing of individual operations and data. This is only a subset of the unit testing for object-oriented software, as the meaning and behaviour of the resources encapsulated in the classes depend on other resources with which they are encapsulated. Integration testing (for conventional software) is truly the unit testing of object-oriented software.

The object-oriented paradigm lowers the complexity and hence, lessens the need for basis path testing. In a nutshell, there are differences between how a conventional software and an object-oriented software should be tested.

16.2.2 Object-Oriented Testing and Maintenance Problems

It is clear from the above discussion that it is not easy to understand and test the OO software. Moreover, it also becomes difficult to maintain it. David *et al.* [68] have recognized some of these problems which are discussed here:

Understanding problem The encapsulation and information hiding features are not easily understood the very first time. There may be a chain of invocations of member functions due to these features. A tester needs to understand the sequence of this chain of invocations and design the test cases accordingly.

Dependency problem There are many concepts in OO that gives rise to the dependency problem. The dependency problem is mainly due to complex relationships, e.g. inheritance, aggregation, association, template class instantiation, class nesting, dynamic object creation, member function invocation, polymorphism, and dynamic binding relationships. These relationships imply that one class inevitably depends on another class. For example, the inheritance relationship implies that a derived class reuses both the data members and the function members of a base class; and hence, it is dependent on the base class.

The complex relationships that exist in an OOS make testing and maintenance extremely difficult in the following way:

(i) Dependency of the classes increases the complexity of understanding and hence testing.

(ii) In OO software, the stub development is more difficult as it needs to understand the chain of called functions, possibly create and properly initialize certain objects, and write their code.

(iii) Template class feature is also difficult to test as it is impossible to predict all the possible uses of a template class and to test them.

(iv) Polymorphism and dynamic binding features add to the challenge of testing them as it is difficult to identify and test the effect of these features.

(v) As the dependencies increase in OOS, it becomes difficult to trace the impact of changes, increasing the chances of bugs, and hence making testing difficult.

State behaviour problem Objects have states and state-dependent behaviours. That is, the effect of an operation on an object also depends on the state of the object and may change its state. Thus, combined effect of the operations must be tested.

16.2.3 Issues in OO Testing

Based on the problems discussed for testing an OO software, the following are some issues which come up while performing testing on OO software.

Basic unit for testing There is nearly universal agreement that class is the natural unit for test case design. Other units for testing are aggregations of classes: class clusters, and application systems. The intended use of a class implies different test requirements, e.g., application-specific vs general-purpose classes, abstract classes, and parameterized (template) classes. Testing a class instance (an object) can verify a class in isolation. However, when verified classes are used to create objects in an application system, the entire system must be tested as a whole before it can be considered verified.

Implications of inheritance In case of inheritance, the inherited features require retesting because these features are now in, new context of usage. Therefore, while planning to test inheritance, include all inherited features. Moreover, planning should also consider the multiple contexts of usage in case of multiple inheritances.

Polymorphism In case of polymorphism, each possible binding of a polymorphic component requires a separate test to validate all of them. However, it may be hard to find all such bindings. This again increases the chance of errors and poses an obstacle in reaching the coverage goals.

White-box testing There is also an issue regarding the testing techniques for OO software. The issue is that the conventional white-box techniques cannot be adapted for OO software. In OO software, class is the major component to be tested, but conventional white-box testing techniques cannot be applied on testing a class. These techniques may be applicable to individual methods but not suitable for OO classes.

Black-box testing The black-box testing techniques may be suitable to OO software for conventional software, but sometimes re-testing of inherited features require examination of class structure. This problem also limits the black-box methods for OO software.

Integration strategies Integration testing is another issue which requires attention and there is need to devise newer methods for it. However, there is no obvious hierarchy of methods or classes. Therefore, conventional methods of incremental integration testing cannot be adopted.

16.2.4 Strategy and Tactics of Testing OOS

The strategy for testing OOS is the same as the one adopted earlier for procedural software, i.e. verification and validation. However, tactics for testing may differ depending upon the structure of OOS.

Object-oriented software testing is generally done bottom-up at four levels:

Method-level testing It refers to the internal testing of an individual method in a class. Existing testing techniques for conventional programming languages are fully applicable to method-level testing.

Class-level testing Methods and attributes make up a class. Class-level (or intra-class) testing refers to the testing of interactions among the components of an individual class.

Cluster-level testing Cooperating classes of objects make up a cluster. Cluster-level (or inter-class) testing refers to the testing of interactions among objects.

System-level testing Clusters make up a system. System-level testing is concerned with the external inputs and outputs visible to the users of a system.

16.2.5 Verification of OOS

The verification process is different in analysis and design steps. Since OOS is largely dependent on OOA and OOD models, it becomes important to verify them. For example, if there is problem in the definition of class attributes and is uncovered during analysis, then this problem will propagate in design and other phases too.

OOA is not matured enough in one iteration only. So it requires many iterations for a reliable analysis model. Verification can play an important role in these iterations of OOA. Each iteration should be verified so that we know that the analysis is complete. For example, consider a class in which a number of attributes are defined during the first iteration, and an attribute may be appended in another iteration due to some misunderstanding. If there is verification of OOA at each iteration, then this error may be pointed out in a review and thereby avoid problems. Thus, all OO models must be tested for their correctness, completeness, etc.

Verification of OOA and OOD models

- Assess the class-responsibility-collaborator (CRC) model and object-relationship diagram.
- Review system design (examine the object-behaviour model to check mapping of system behaviour to subsystems, review concurrency and task allocation, use use-case scenarios to exercise user interface design).
- Test object model against the object relationship network to ensure that all design objects contain necessary attributes and operations needed to implement the collaborations defined for each CRC card.
- Review detailed specifications of algorithms used to implement operations using conventional inspection techniques.

Verification points of some of the models are given below:

- Check that each model is correct in the sense that proper symbology and modeling conventions have been used.
- Check that each use-case is feasible in the system and unambiguous such that it should not repeat operation.
- Check that a specific use-case needs modification in the sense that it should be divisible further in more use-cases.
- Check that actors identified in use-case model are able to cover the functionality of the whole system. If any new actor is needed to fulfill the requirement, then add it.

- All interfaces between actors and use-cases in use-case diagram must be continuously checked such that there should be no inconsistency.
- All events and their sequences must be checked iteratively in a sequence diagram. If any detail of the operation is missing, then events can be added or deleted if required.
- Collaboration diagrams must be verified for distribution of processing between objects. Suppose the collaboration diagram was shaped like a star, with several objects communicating with a central object. Analyst may conclude that the system is too dependent on the central object and redesign the objects to distinguish the processing power more evenly [37].
- Each identified class must be checked against defined selection characteristics such that it should be unique and unambiguous.
- All collaborations in class diagrams must be verified continuously such that there should be no inconsistency.
- Check the description of each CRC index card to determine the delegated responsibility.

16.2.6 Validation Activities

All validation activities described earlier are also implemented in OOS. But the meaning of unit and integration testing changes due to the nature of OOS. Therefore, new methods or techniques are required to test all the features of OOS.

Unit/Class Testing

Unit or module testing has no meaning in the context of OOS. Since the structure of OOS is not based on action, module testing has lost its meaning. OOS is based on the concept of class and objects. Therefore, the smallest unit to be tested is a class. The nature of OOS is different as compared to procedural software; therefore many issues rise for testing a class. These issues are discussed in the next section.

Issues in testing a class

- A class cannot be tested directly. First, we need to create the instances of a class as object and then test it. Thus, an abstract class cannot be tested, as its objects cannot be created.
- Operations or methods in a class are the same as functions in traditional procedural software. One idea is that they can be tested independently using the previous white-box or black-box methods. But in the context

of OOS, this isolated testing has no meaning. Thus the methods should also be tested with reference to a class or class hierarchy in the direction of complete testing of a class.

- One important issue in class testing is the testing of inheritance. If some classes have been derived from a base class, then independent base class testing is not sufficient. All derived classes in the hierarchy must also be tested depending on the kind of inheritance. Thus, the following special cases also apply to inheritance testing:
  (a) Abstract classes can only be tested indirectly (because it can't be instantiated). Hence, abstract classes have to be tested by testing instance of its descendant specialized classes.
  (b) Special tests have to be written to test the inheritance mechanism, e.g. a child class can inherit from a parent and that we can create instance of the child class.
- There may be operations in a class that change the state of the object on their execution. Thus, these operations should be tested with reference to state change of object.
- Polymorphism provides us with elegant compact code, but it can also make testing more complicated. The identification of all the paths through a section of code that uses polymorphism is harder because the criteria which indicate different paths are not specifically mentioned in the code.

16.2.7 Testing of OO classes

Feature-based Testing of Classes

There are a number of code complexity metrics available which vary from simple ones such as the number of the source lines of code, to the McCabe cyclomatic complexity metric. These metrics can be used to measure the complexity of the individual features of a class providing the tester with an accurate way of determining which features to consider first for testing. The features of a class can be categorized into six main groups [69, 70]:

Create These are also known as *constructors*. These features perform the initial memory allocation to the object, and initialize it to a known state.

Destroy These are also known as *destructors*. These features perform the final memory de-allocation when the object is no longer required.

Modifiers The features in this category alter the current state of the object, or any part thereof.

Predicates The features in this category test the current state of the object for a specific instance. Usually, they return a BOOLEAN value.

Selectors The features in this category examine and return the current state of the object, or any part thereof.

Iterators The features in this group are used to allow all required sub-objects to be visited in a defined order.

Testing feature groups While testing the classes with features as discussed above, there may be some preferences of these feature groups according to which one is tested first. Turner and Robson [71] have provided the following guidelines for the preferred features to be tested:

(i) The *create* features of a class must be verified first, as these features are for the initialization of every object created. Therefore, they must be tested rigorously.

(ii) The *selector* features can be the next features to be verified as they do not alter the state of the object. To test them, alter the state of the object either by using the appropriate constructor (if available), or by directly altering the state of the object.

(iii) The *predicate* features of a class test for specific states of the object and therefore while constructing the test cases, care must be taken to place the object in the specific state for the required test result.

(iv) The *modifier* features of a class are used to alter the state of the object from one valid state to another. To detect if the transition was to the correct state, the selectors and predicate features should be used where possible.

(v) The *destructor* feature of a class performs any 'housekeeping', for example, returning any allocated memory to the heap. To test them, features which alter the state of the object and allocate memory are used to create the scenarios for the test cases, then the destructor is called.

Role of Invariants in Class Testing

If we are able to specify the valid states and in valid states for an object, it can help in testing the classes. The language Eiffel provides the facility of specifying the invariants. But, by providing the features for testing the validity of the object's current state, you can also achieve this ability. Turner and Robson [71] have also shown the importance of invariants in class testing.

For example, take a class to implement an integer Stack as:
Class Stack with following data members:

ItemCount, empty, full, push, pop, max_size

Here in this class, invariants can be defined as given below:

1. ItemCount cannot have a negative value.
2. There is an upper bound check on ItemCount, i.e. ItemCount ≤ max-size
3. Empty stack, i.e. ItemCount = 0
4. Full Stack, i.e. ItemCount = max_size

All invariants discussed above show the states of the class. These invariants allow the validity of the current state of the object to be tested easily. An invariant is also useful for ensuring that the data representation is in a valid state, even if it has been directly altered. This can help maintain the integrity of the class.

State-Based Testing

State-based testing can also be adopted for testing of object-oriented programs. The systems can be represented naturally with finite state machines as states correspond to certain values of the attributes and transitions correspond to methods. The features of a class provide the desired behaviour by interacting with the data-representation. Although these interactions are found in programs written in traditional languages, they are more visible and prevalent in object-oriented programming languages. State-based testing tests these interactions by monitoring the effects that features have on the state of an object. The state can be determined by the values stored in each of the variables that constitute its data representation. Each separate variable is known as a *substate*, and the values it may store are known as *substate-values*.

The restrictions on objects that are to be represented as state machines include [71]:

- The behaviour of the object can be defined with a finite number of states.
- There are a finite number of transitions between the states.
- All states are reachable from the initial state.
- Each state is uniquely identifiable through a set of methods.

State-based testing based on the *finite state automata*, tests each feature with all its valid input states. However, it may not be possible to use every single value for each variable of the data-representation due to their potentially large

range. Therefore, the categorization is applied in order for a suitable reduction to be achieved.

For designing the test cases, we need to have stimuli. Stimuli are particular values, obtained after analysing the feature's parameters, and have specific effects on the response of the feature, or are significant. The test cases then consist of applying each stimulus in combination with each possible state for the representation. The resultant states generated are then validated against those expected. The behaviour of an object can be predicted by matching the input states expected by one feature with the output states of another. This validation should ensure the correct operation of the class with no concern for the order in which the features are invoked.

Process of state-based testing The process of using state-based testing techniques is as follows:

1. Defines the states.
2. Define the transitions between states.
3. Define test scenarios.
4. Define test values for each state.

To see how this works, consider the Stack example again.

Define states The first step in using state-based testing is to define the states. We can create a state model with the following states in the Stack example:

1. Initial: before creation
2. Empty: ItemCount = 0
3. Holding: ItemCount > 0, but less than the max capacity
4. Full: ItemCount = max
5. Final: after destruction

Define transitions between states The next step is to define the possible transitions between states and to determine what triggers a transition from one state to another. Generally, when we test a class, a transition is triggered when a method is invoked. Thus, some possible transitions could be defined as follows:

1. **Initial → Empty:**

   Action = "create" e.g. "s = new Stack()" in Java
2. **Empty → Holding:**

   Action = "add"
3. **Empty → Full:**

Action = "add" if max_size= 1

4. **Empty → Final:**

   Action = "destroy" e.g. destructor call in C++, garbage collection in Java

5. **Holding → Empty:**

   Action = "delete"

Thus, for each identified state, we should list the following:

1. Valid actions (transitions) and their effects, i.e., whether or not there is a change of state.
2. Invalid actions that should produce an exception.

Define test scenario Tests generally consist of scenarios that exercise the object along a given path through the state machine. The following are some guidelines:

1. Cover all identified states at least once.
2. Cover all valid transitions at least once.
3. Trigger all invalid transitions at least once.

Test cases are designed such that they exercise operations that change state and those that do not change state separately. For instance,

Test case: Create-Add-Add-Delete-Add-delete-destroy

It is also needed to check the state of the object that is being tested throughout the scenario to ensure that the theoretical state model you have defined is actually the one implemented by the class you are testing.

Define test values for each state Finally, choose test values for each individual state. The consideration in choosing the values is that choose unique test values and do not reuse values that were used previously in the context of other test cases. This strategy provides more diversity in the test suite and increases the likelihood of detecting bugs.

16.2.8 Inheritance Testing

The issues in testing the inheritance feature are discussed below.

Superclass modifications When a method is changed in superclass, the following should be considered:

(i) The changed method and all its interactions with changed and unchanged methods must be retested in the superclass.

(ii) The method must be retested in all the subclasses inheriting the method as extension.

(iii) The 'super' references to the method must be retested in subclasses.

Inherited methods The following inherited methods should be considered for testing:

(i) *Extension:* It is the inclusion of superclass features in a subclass, inheriting method implementation and interface (name and arguments).

(ii) *Overriding:* It is the new implementation of a subclass method, with the same inherited interface as that of a superclass method.

Reusing of test suite of superclass It may not be possible to reuse superclass tests for extended and overridden methods due to the following reasons:

(i) Incorrect initialization of superclass attributes by the subclass.

(ii) Missing overriding methods.

(iii) Direct access to superclass fields from the subclass code can create a subtle side.

(iv) A subclass may violate an invariant from the superclass or create an invalid state.

Addition of subclass method When an entirely new method is added to a specialized subclass, the following should be considered:

(i) The new method must be tested with method-specific test cases.

(ii) Interactions between the new method and the old methods must be tested with new test cases.

Change to an abstract superclass interface 'Abstract' superclass is the class wherein some of the methods have been left without implementation; just the interface has been defined. In this case, all the subclasses must be retested, even if they have not changed.

Interaction among methods To what extent should we exercise interaction among methods of all superclasses and of the subclass under test?

Thus, inherited methods should be retested in the context of a subclass.

Example: If we change some method m() in a superclass, we need to retest m() inside all subclasses that inherit it.

Example: If we add or change a subclass, we need to retest all methods inherited from a superclass in the context of the new/changed subclass.

Inheritance of Invariants of Base Class

If an invariant is declared for a base class, the invariant may be extended by the derived class. Additional clauses may be added for new data-members

declared by the derived class. However, the data-representation of the parent class still complies with the original invariant and therefore does not require retesting.

If the invariant provided by the base class is retracted, then the base class must be retested with respect to the new invariant. A retraction involves the loosening of clauses, allowing the data members of the parent class to accept states that were previously not allowed. The retesting can be performed by re-executing all the original test cases, along with some additional test cases to cover the widened invariant. By clever analysis of the widened invariant, it may be possible to reduce the number of test cases from the original suite of test cases that need to be re-executed.

Incremental Testing

Wegner and Zdonik [73] proposed this technique to test inheritance. The technique first defines inheritance with an incremental hierarchy structure. Then this approach is used to test the inheritance of classes. This technique is based on the approach of a modifier to define a derived class. A *modifier* is first applied on the base class that alters its attributes. The modifier and the base class, along with the inheritance rules for the language, are used to define the derived class. We use the composition operator $\oplus$ to represent this uniting of base class and modifier, and we get

$$D = B \oplus M$$

where D is derived class, B is base class, and M is a modifier. This can be seen in Fig. 16.1.

Figure 16.1 Deriving class D from base class B using modifier M

The derived class designer specifies the modifier, which may contain various types of attributes that alter the base class to get the derived class. The following attributes have been identified:

- New attribute
- Inherited attribute
- Redefined attribute
- Virtual-new attribute

- Virtual-inherited attribute
- Virtual-redefined attribute

All the above mentioned attributes have been illustrated in the Fig. 16.2.

After defining the inheritance as an incremental hierarchical approach, now we will use this technique to test the inheritance. This is known as the incremental class technique for testing inheritance. To test the derived class, the issue is to find out which attributes and interactions between them need to be tested. This can be managed easily if we divide it into two categories: *Testing of features* (TF) and *testing of interactions* (TI). With this information for every class, the idea of incremental testing is that initially, the base class will be tested by testing each feature and interactions between them. But, their execution is maintained in a testing history of test cases. This testing history will be reused for the features which are being inherited without any modification and unchanged context, thereby reducing the effort being made in testing everything in the inherited subclasses. Only new features or those inherited-affected features and their interactions are tested as added efforts. The following steps describe the technique for incremental testing:

1. Test the base class by testing each member function individually and then testing the interactions among the member functions. All the testing information is saved in a testing history. Let it be *TH*.
2. For every derived class, determine the modifier *m* and make a set *M* of all features for all derived classes.
3. For every feature in *M*, look for the following conditions and perform testing:
   (i) If feature is new or virtual-new, then test the features and the interactions among them as well.
   (ii) If feature is inherited or virtual-inherited without any affect, then test only the interactions among the features. If there is a need to test the features again in the derived class, then the test history *TH* of base class can be reused for the test cases.
   (iii) If feature is redefined or virtual-redefined, then test the features and the interactions among them as well. For testing, use the test cases from testing history *TH* of the base class. However, if the redefined feature is affected somewhere or there is need to interact with new attribute, new test cases should be designed.

The benefit of this technique is that we don't need to analyse the derived classes for what features must be tested and what may not be tested, thereby reducing the time required in analysing and executing the test cases.

```
Class B
{
    float x;
    float y;
public:
    B() {}
    void method1(
float a, float b)
    {
    x=a; y=b;
    }

virtual float method2()
    {
    return x;
    }

    float method3()
    {
    return y;
    }

    float method4()
    {
    return method2();
    }

virtual float method5()
    {
    return x;
    }
};
```

```
Class D: public B
{
private:
int i;
public:
   D() {}
   void method1(int a)
     {
     i = a + 5;
     }

virtual float method2()
{
return 4*B::method2();
}

float method3()
{
return 6*B::method3();
}

virtual int method6()
{
return i;
}
};
```

```
D's attributes after the mapping

private:
int i;  //New
public:
D() {}  //New

void method1(float a, float b)
//Inherited
    {
    x=a; y=b;
    }

    void method1(int a) //New
    { i = a + 5; }

virtual float method2()
// virtual redefined
{
return 4*B::method2();
}

float method3() //redfined
{
return 6*B::method3();
}

float method4() //inherited
{
return method2();
}

virtual float method5()
//virtual inherited
{
return x;
}

virtual int method6()
//virtual new
{
return i;
}
```

Figure 16.2 Incremental modification technique

16.2.9 Integration Testing

Different classifications of testing levels have been proposed with the following terminology:

Inter-class testing This type of testing includes testing of any set of cooperating classes, aimed at verifying that the interaction between them is correct. There are no constraints on how these classes are selected.

Cluster testing This type of testing includes testing of the interactions between the different classes belonging to a subset of the system, having some specific properties called a *cluster*. Usually, a cluster is composed of cooperating classes providing particular functionalities (e.g., all the classes which can be used to access the file-system, or the classes composing a Java package). Clusters should provide a well-defined interface, i.e. their interfaces should be well-understood and they should mutually interact only by means of their interfaces.

Inter-cluster testing This type of testing includes testing of the interactions between already tested clusters. The result of the integration of all clusters is the whole system.

Thread-based Integration Testing

As long as we consider object-oriented systems as sets of cooperating entities exchanging messages, threads can be naturally identified with sequences of subsequent message invocations. Therefore, a thread can be seen as a scenario of normal usage of an object-oriented system. Testing a thread implies testing the interactions between classes according to a specific sequence of method invocations. This kind of technique has been applied by several authors. Kirani and Tsai [77] propose a technique for generating test cases from functional specification for module and integration testing of object-oriented systems. The method aims at generating test cases that exercise specific combinations of method invocations and provides information on how to choose classes to be integrated.

A similar approach is proposed by Jorgensen and Erickson [78], who introduce the notion of *method-message path* (*MM-path*), defined as a sequence of method executions linked by messages. For each identified MM-path, integration is performed by pulling together classes involved in the path and exercising the corresponding message sequence. Thus, integration level thread is a MM-path. More precisely, Jorgensen and Erickson identify two different levels for integration testing:

Message quiescence This level involves testing a method together with all methods it invokes, either directly or transitively.

Event quiescence This level is analogous to the message quiescence level, with the difference that it is driven by system-level events. Testing at this level

means exercising message chains (threads), such that the invocation of the first message of the chain is generated at the system interface level (i.e., the user interface) and, analogously, the invocation of the last message results in an event which can be observed at the system interface level. An end-to-end thread is called an *atomic system function* (*ASF*). Therefore, a system-level thread is a sequence of ASFs. ASFs have port events as their inputs and outputs, and thus a sequence of ASFs implies an interleaved sequence of port input and output events. The main drawback of this method is the difficulty in the identification of ASFs, which requires either the understanding of the whole system or an analysis of the source code.

Implicit Control Flow-based Integration Testing

The integration testing should also consider the methods invocations where the call is not obvious in the source code. For example, the C++ expression "x*y" for object *x* and *y*. In this case, there is a call to a method *via* operator overloading for the multiplication of two objects. Thus, these types of method invocations also need to be considered in integration testing. This is known as implicit control flow-based integration testing [74].

There are three approaches for this type of integration: *optimistic, pessimistic, and balanced.* To understand all three approaches, let us take an example as shown in Fig. 16.3. This example illustrates the very common example of object-oriented feature where we want to calculate the salary of employees of many categories, e.g. worker, assistant, and manager using the common method CalSalary(). The CalSalary() will be run for which employee is known at run time, i.e. it is a dynamically bound method. In this example, assume that employee is a base class; worker, assistant, and manager are three derived classes from employee. Each derived class uses the method Calsalary() with different computations. But at run time, it will be decided which CalSalary() method is invoked. Thus, method invocations are not obvious. In this case, we will use the following three integration approaches as discussed below.

Figure 16.3 Implicit control flow example

Optimistic approach This approach is based on the optimistic view that method invocation will be confined to the level of abstraction of the source code only. Therefore, in this type of integration we will test only those method invocation or interfaces which support this abstraction as shown in Fig. 16.4.

Figure 16.4 Optimistic approach to implicit control flow

Pessimistic approach This approach assumes that every class can call the common method CalSalary. Therefore, all interfaces of calling the method are tested here as shown in Fig. 16.5. However, the danger in this type of integration testing is the increase in the complexity in testing.

Figure 16.5 Pessimistic approach to implicit control flow

Balanced approach This approach is a balance between the above two approaches. We must take the interfaces according to the code as well as expected method invocations. But, here we take a mid approach that all combinations of interfaces in the pessimistic approach need not be tested. Instead, only one or two classes should test source and some combinations of interfaces will be tested such that other interfaces are likely to work properly. Therefore, we can make an equivalence set of interfaces which will be tested and others will be assumed to be working correctly as shown in Fig. 16.6.

16.2.10 UML-Based OO Testing

UML is a design and modeling language for object-oriented software and is widely used as a *de facto* standard that provides techniques to model the software from different perspectives. It supports facilities both for abstract high-

level modeling and for more detailed low-level modeling. It consists of a variety of graphical diagram types that can be extended for specific application areas. It is associated with a formal textual language, *OCL* (Object Constraint Language). UML also provides support for model-based testing.

Figure 16.6 Balanced approach to implicit control flow

UML diagrams in software testing

The following UML diagrams are helpful in the testing activities mentioned below:

- *Use-case diagrams*: testing of system-level functional requirements, acceptance testing
- *Class diagrams*: class (module / unit) testing, integration testing
- *Sequence diagrams, collaboration diagrams*: integration testing, testing of control and interaction between objects, testing of communication protocols between (distributed) objects
- *Activity diagrams*: testing of work flow and synchronization within the system, white-box testing of control flow
- *State diagrams* (state charts): state-based testing
- *Package diagrams*, *component diagrams*: integration testing
- *Deployment diagrams*: system testing

System Testing based on Use-Cases

Use-case In general, a use-case is a sequence of interactions by which the user accomplishes a task in a dialogue with the system. When we use use-case in testing, we follow the following:

- use-case = one particular way to use the system
- use-case (in testing) = user requirement
- set of all use-cases = complete functionality of the system
- set of all use-cases (in testing) = interface between the users (*actors*) and the system

Scenario It is an instance of a use-case, expressing a specific task, by a specific actor, at a specific time, and using specific data.

Use-case model This is a set of use-case diagrams, each associated with a textual description of the user's task. For testing, the use-case model must be extended with:

- the domain of each variable participating in the use-cases,
- the input/output relationships among the variables,
- the relative frequency of the use-cases,
- the sequential (partial) order among the use-cases.

Thus, use, case-based testing is a technique for generating test cases and recommended configurations for system-level testing. The testers build a test model based on the standard UML notions of use-cases, actors, and the relationships between these elements. The use-cases are enhanced with additional information, including inputs from actors, outputs to the actors, and how the use-case affects the state of the system.

To perform this testing, the tester needs to identify four things:

- the use-cases of interest,
- the actors involved in using the system,
- the input, output, and system effects for the use-cases,
- the flows of interest between the use-cases.

A use-case is a semantically meaningful function that provides some value from the user's point of view. For example, saving a file in a word processing system would be represented by the Save File use-case. Each use-case can have input parameters associated with it, and for each parameter, a set of logical partitions of the values that parameter can take may be identified. Finally, the use-cases can be connected using flows that describe a sequence of use-case that are performed to accomplish some goal.

SUMMARY

Object-oriented software is different in nature from procedural software. Its origin is based on reducing the complexity of the software development. However, this type of environment adds many issues that need to be resolved while testing it. This chapter discusses the issues regarding OO technology and techniques for testing the OO software. The meaning of unit is not the same as seen in the procedural software. Similarly, there is no obvious hierarchy as observed in procedural software; therefore top-down and bottom-up integration testing cannot be adopted straightaway. Inheritance is another issue which needs attention while testing, otherwise we will end up with heavy testing of base class and derived classes. The techniques to deal with the unit, integration, and system testing have been discussed.

The UML model is another big support for testing the OO software. Since OO software can be developed with the help of UML models in the form of graphical tools, these also help in testing the OO software at various levels. The chapter also discusses them and elaborates system testing with use-case model.

Let us review the important concepts described in this chapter:

- Object-oriented software is not easy to test as it is different in nature from procedural software.
- The encapsulation and information hiding features are not easily understood the very first time.
- There is dependency problem in OOS due to complex relationships, e.g. inheritance, aggregation, association, template class instantiation, class nesting, dynamic object creation, member function invocation, polymorphism, and dynamic binding relationships. These relationships imply that one class inevitably depends on another class.
- The natural unit for test case design in OOS is a class.
- The conventional white-box testing techniques cannot be applied on testing a class.
- Object-oriented software testing is generally done bottom-up at four levels: Class-level, method-level, cluster-level, and system-level.
- Since OOS is largely dependent upon OOA and OOD models, it becomes important to verify them. Thus, the verification of OOS is largely to verify OOA and OOD models.
- There are features of every class which can be prioritized for testing OO classes. This is known as feature-based testing of OO classes.
- State-based testing can also be adopted for the testing of object-oriented programs.
- Incremental testing is the technique to test the inheritance wherein inheritance is defined as an incremental hierarchy structure.
- Integration testing of OOS is done at the following levels: inter-class level, cluster-level, and inter-cluster level.
- Thread-based testing is the integration testing method wherein testing a thread implies testing interactions between classes according to a specific sequence of method invocations.
- Implicit control flow-based testing is the integration testing wherein we consider methods invocations where the call is not obvious in the source code. There are three approaches for this type of integration: optimistic, pessimistic, and balanced.
- Use, case-based testing is a technique for generating test cases and recommended configurations for system, level testing. The testers build a test model based on the standard UML notions of use-cases, actors, and the relationships between these elements.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Encapsulation is _____.
   (a) the ability to derive new class from existing classes

(b) avoiding the accidental change in data such that no one can directly access the object's data
(c) having many forms and implementations of a particular functionality
(d) none of the above

2. Inheritance is _____.
   (a) the ability to derive new class from existing classes
   (b) avoiding the accidental change in data such that no one can directly access the object's data
   (c) having many forms and implementations of a particular functionality
   (d) none of the above
3. Polymorphism is _____.
   (a) the ability to derive new class from existing classes
   (b) avoiding the accidental change in data such that no one can directly access the object's data
   (c) having many forms and implementations of a particular functionality
   (d) none of the above
4. A use-case is a _____.
   (a) scenario
   (b) test case
   (c) an actor
   (d) all of the above
5. Use-case diagrams show the _____.
   (a) visual interaction between use-cases and test cases
   (b) visual interaction between use-cases and classes
   (c) visual interaction between use-cases and objects
   (d) visual interaction between use-cases and actors
6. Dependency problem is caused by _____.
   (a) encapsulation and information hiding features
   (b) state dependent behaviours
   (c) the complex relationships that exist in an OOS
   (d) none of the above
7. Understanding problem is caused by _____.
   (a) encapsulation and information hiding features
   (b) state dependent behaviours
   (c) the complex relationships that exist in an OOS
   (d) none of the above

8. State behaviour problem is caused by _____.
   (a) encapsulation and information hiding features
   (b) state dependent behaviours
   (c) the complex relationships that exist in an OOS
   (d) none of the above
9. Method-level (or unit) testing refers to _____.
   (a) testing of interactions among the components of an individual class
   (b) the internal testing of an individual method in a class
   (c) testing of interactions among objects
   (d) external inputs and outputs visible to the users of a system
10. Class-level testing refers to _____.
   (a) testing of interactions among the components of an individual class
   (b) the internal testing of an individual method in a class
   (c) testing of interactions among objects
   (d) external inputs and outputs visible to the users of a system
11. Cluster-level testing refers to _____.
   (a) testing of interactions among the components of an individual class
   (b) the internal testing of an individual method in a class
   (c) testing of interactions among objects
   (d) external inputs and outputs visible to the users of a system
12. System-level testing refers to _____.
   (a) testing of interactions among the components of an individual class
   (b) the internal testing of an individual method in a class
   (c) testing of interactions among objects
   (d) external inputs and outputs visible to the users of a system
13. The features of a class can be categorized into _____ main groups.
   (a) 3
   (b) 5
   (c) 2
   (d) 6
14. _____ feature must be tested first in feature-based testing.
   (a) Destroy
   (b) Predicate
   (c) Modifiers
   (d) Create
15. _____ feature is an ideal candidate for use in the validation of other features.
   (a) Destroy

(b) Predicate
(c) Selectors
(d) Create

16. The modifier features of a class are used to _____.
    (a) alter the state of the object from one valid state to another valid state.
    (b) alter the state of the class from one valid state to another valid state.
    (c) all of the above
17. Inherited methods should be retested in the context of a _____.
    (a) superclass
    (b) subclass
    (c) objects
    (d) all of the above
18. System testing can be performed with _____.
    (a) deployment diagrams
    (b) class diagram
    (c) package diagram
    (d) use case diagram
19. Sequence diagrams are used for _____.
    (a) unit testing
    (b) integration testing
    (c) state testing
    (d) none of the above
20. Package diagrams are used for _____.
    (a) unit testing
    (b) integration testing
    (c) state testing
    (d) none of the above

Review Questions

1. What is the major reason for discarding the structured approach?
2. Differentiate an object and a class with example.
3. What is the difference between testing a procedural software and an object-oriented software?
4. What are the testing and maintenance problems introduced with object-oriented software?
5. What steps would you take to verify a use-case model?

6. Take an OOS project and verify its use-case model.
7. What steps would you take to verify a CRC model?
8. Take an OOS project and verify its CRC model.
9. What are the issues in testing a class?
10. Make a software to calculate the salary of employees in an organization, assuming the necessary details required using the object-oriented technology and prepare the following:
    (a) Use-case diagram
    (b) Sequence diagram
    (c) Collaboration diagram
    (d) CRC model
    (e) Class diagram
    (f) Verify use-case model
    (g) Verify CRC model
11. Perform feature-based testing on the classes identified in Question 10.
12. What is the role of invariants in class testing? Discuss with example.
13. List the inheritance issues in testing the classes of Question 10.
14. Identify inheritance relationship in the class identified in Question 10 and show example of every attribute in the incremental testing. Perform incremental testing on the classes mentioning all the attributes.
15. What are the integration testing levels of an OOS?
16. What is the procedure for performing thread-based integration testing?
17. Perform thread-based integration on the classes identified in Question 10.
18. What is the idea behind implicit control flow integration testing?
19. Is it possible to perform implicit control flow integration testing on the classes identified in Question 10.
20. Which UML diagrams are helpful in testing an OOS?
21. Perform system testing based on the use-cases identified in use-case model prepared in Question 10.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Chapter 17

Testing Web-based Systems

OBJECTIVES

After reading this chapter, you should be able to understand:

- How the testing of a web-based system is different from testing traditional software
- Challenges in testing web-based systems
- Web engineering based analysis and design models which is the basis for testing web-based systems
- Interface testing
- Usability testing
- Content testing
- Navigation testing
- Configuration/Compatibility testing
- Security testing
- Performance testing
- Load testing
- Stress testing

Web-based systems have had a great impact on our daily lives. These systems have evolved from small website add-ons to large multi-tiered applications. The computing environment for this rapidly growing web technology is complex as Internet is heterogeneous, distributed, and multi-platform. Besides the dynamic computing environment for web-based systems, user requirements for new features also add to the complexity in designing and testing these systems.

The growing number of web applications combined with an ever-growing Internet user mass, emphasizes the importance of developing high-quality products. However, many attributes of quality web-based systems such as ease of navigation, accessibility, scalability, maintainability, usability, compatibility and interoperability, security, readability, and reliability are not given due consideration during development. Therefore, proper testing of web-based systems is needed in ensuring reliable, robust, and high-performing operation of web applications.

The testing of web-based systems is a complex task not only due to the overwhelming number of users on the web but there is also a lot of difference between traditional systems and the web-based systems. Traditional testing techniques are not adequate for web-based systems because they do not incorporate the problems associated with the computation environment, new features, and constraints of web-based applications.

17.1 Web-based System

The web-based software system consists of a set of web pages and components that interact to form a system which executes using web server(s), network,

HTTP, and a browser, and in which user input affects the state of the system. Thus, web-based systems are typical software programs that operate on the Internet, interacting with the user through an Internet browser. Some other terms regarding these systems are given below:

Web page is the information that can be viewed in a single browser window.

A *website* is a collection of web pages and associated software components that are related semantically by content and syntactically through links and other control mechanisms. Websites can be dynamic and interactive.

A *web application* is a program that runs as a whole or in part on one or more web servers and that can be run by users through a website. Web applications require the presence of web server in simple configurations and multiple servers in more complex settings. Such applications are called *web-based applications*. Similar applications, which may operate independent of any servers and rely on operating system services to perform their functions, are termed *web-enabled applications*. These days, with the integration of technologies used for the development of such applications, there is a thin line separating *web-based* and *web-enabled applications*, so we collectively refer to both of them as *web applications*.

17.2 Web Technology Evolution

17.2.1 First Generation/ 2-tier Web system

The web systems initially were based on a typical client-server model. The *client* is a web browser that people use to visit websites. The websites are on different computers, the *servers* and the HTML files are sent to the client by a software package called a *web server*. HTML files contain JavaScripts, which are small pieces of code that are interpreted on the client. HTML forms generate data that are sent back to the server to be processed by CGI programs. This 2-tier architecture consisting of two separate computers was a simple model suitable for small websites but with little security.

17.2.2 Modern 3-tier and N-tier Architecture

In the 2-tier architecture, it was difficult to separate presentation from business logic with the growth of websites. Due to this, applications were not scalable and maintainable. Moreover, having only one web server imposes a bottleneck; if there is a problem on that server, then the users cannot access the website (availability). Therefore, this simple client-server model was expanded first to the 3-tier model and then, to the N-tier model. To get quality

attributes, such as security, reliability, availability, scalability, and functionality, application server has been introduced wherein most of the software has been moved to a separate computer. Indeed, on large websites, the application server is actually a collection of application servers that operate in parallel. The application servers typically interact with one or more *database servers*, often running a commercial database (Fig. 17.1). Web and application servers often are connected by *middleware*, which are packages provided by software vendors to handle communication, data translation, and process distribution. Likewise, the application-database servers often interact through middleware. The web server that implements CGI, PHP, Java Servlets, or Active Server Pages (ASP), along with the application server that interacts with the database and other web objects is considered the middle tier. Finally, the database along with the DBMS server forms the third tier.

Figure 17.1 Modern websites

In the N-tier model, there are additional layers of security between potential crackers and the data and application business logic. Separating the presentation (typically on the web server tier) from the business logic (on the application server tier) makes it easier to maintain and expand the software both in terms of customers that can be serviced and services that can be offered. The use of distributed computing, particularly for the application servers, allows the web application to tolerate failures, handle more customers, and allows developers to simplify the software design.

17.3 Traditional Software and Web-based Software

Web systems are based on client-server architecture wherein a client typically enables users to communicate with the server. Therefore, these systems share some characteristics of client-server architecture. However, there are a number of aspects of web systems that necessitate having different techniques to test them. These are discussed below [147]:

1. Clients of the traditional client-server systems are platform-specific. This means that a client application is developed and tested for each supported

client operating system. But the web client is operating within the web browser's environment. Web browsers already consist of operating system-specific client software running on a client computer. But these browsers need to support HTML, as well as active contents to display web page information. For this purpose, browser vendors must create rendering engines and interpreters to translate and format HTML contents. In making these software components, various browsers and their releases introduce incompatibility issues.

2. Web-based systems have a more dynamic environment as compared to traditional client-server systems. In client-server systems, the roles of the clients and servers and their interactions are predefined and static as compared to web applications where client side programs and contents may be generated dynamically. Moreover, the environment for web applications is not predefined and is changing dynamically i.e. hardware and software are changing, configuration are ever-changing, etc. Web applications often are affected by these factors that may cause incompatibility and interoperability issues.
3. In the traditional client-server systems, the normal flow of control is not affected by the user. But in web applications, users can break the normal control flow. For example, users can press the back or refresh button in the web browser.
4. Due to the dynamic environment, web systems demand more frequent maintenance.
5. The user profile for web systems is very diverse as compared to client-server systems. Therefore, the load on web access due to this diversity is not predictable.

17.4 Challenges in Testing for Web-based Software

As discussed above, web-based software are different as compared to traditional systems. Therefore, we must understand the environment of their creation and then test them. Keeping in view the environment and behaviour of web-based systems, we face many challenges while testing them. These challenges become issues and guidelines when we perform testing of these systems. Some of the challenges and quality issues for web-based system are discussed here:

Diversity and complexity Web applications interact with many components that run on diverse hardware and software platforms. They are written in diverse languages and they are based on different programming approaches such as procedural, OO, and hybrid languages such as Java Server Pages

(JSPs). The client side includes browsers, HTML, embedded scripting languages, and applets. The server side includes CGI, JSPs, Java Servlets, and .NET technologies. They all interact with diverse back-end engines and other components that are found on the web server or other servers.

Dynamic environment The key aspect of web applications is its dynamic nature. The dynamic aspects are caused by uncertainty in the program behaviour, changes in application requirements, rapidly evolving web technology itself, and other factors. The dynamic nature of web software creates challenges for the analysis, testing, and maintenance for these systems. For example, it is difficult to determine statically the application's control flow because the control flow is highly dependent on user input and sometimes in terms of trends in user behaviour over time or user location. Not knowing which page an application is likely to display hinders statically modeling the control flow with accuracy and efficiency.

Very short development time Clients of web-based systems impose very short development time, compared to other software or information systems projects. For e.g. an e-business system, sports website, etc.

Continuous evolution Demand for more functionality and capacity after the system has been designed and deployed to meet the extended scope and demands, i.e. scalability issues.

Compatibility and interoperability As discussed above, there may also be compatibility issues that make web testing a difficult task. Web applications often are affected by factors that may cause incompatibility and interoperability issues. The problem of incompatibility may exist on both the client as well as the server side. The server components can be distributed to different operating systems. Various versions of browsers running under a variety of operating systems can be there on the client side. Graphics and other objects on a website have to be tested on multiple browsers. If more than one browser will be supported, then the graphics have to be visually checked for differences in the physical appearance. The code that executes from the browser also has to be tested. There are different versions of HTML. They are similar in some ways but they have different tags which may produce different features.

17.5 Quality Aspects

Based on the above discussion on the challenges of web software, some of the quality aspects which must be met while designing these systems are discussed here.

Reliability It is an expected feature that the system and servers are available at any time with the correct results. It means that the web systems must have high availability to be a reliable system. If one web software fails, the customer should be able to switch to another one. Thus, if web software is unreliable, websites that depend on the software will lose customers and the businesses may lose large amount of money.

Performance Performance parameters for web systems must also be tested. If they are not tested, they may even lead to the total failure of the system. Performance testing is largely on the basis of load testing, response time, download time of a web page, or transactions performed per unit time.

Security Since web systems are based on Internet, there are possibilities of any type of intrusion in these systems. Thus, security of web applications is another challenge for the web applications developers as well as for the testing team. Security is critical for e-commerce websites.

Tests for security are often broken down into two categories: testing the security of the infrastructure hosting the web application and testing for vulnerabilities of the web application. Some of the points that should be considered for infrastructure are firewalls and port scans. For vulnerabilities, there is user authentication, cookies, etc. Data collected must be secured internally. Users should not be able to browse through the directories in the server. A cookie is a text file on a user's system that identifies the user. Cookies must always be encrypted and must not be available to other users.

Usability This is another major concern while testing a web system. The users are real testers of these systems. If they feel comfortable in using them; then the systems are considered of high quality. Thus, usability also becomes an issue for the testers. The outlook of the system, navigation steps, all must be tested keeping the large categories of users in mind. Moreover, the users expect to be able to use the websites without any training. Thus, the software must flow according to the users' expectations, offer only needed information, and when needed, should provide navigation controls that are clear and obvious.

Scalability With the evolution of fast technology changes, the industry has developed new software languages, new design strategies, and new communication and data transfer protocols, to allow websites to grow as needed. Therefore, designing and building web software applications that can be easily scaled is currently one of the most interesting and important quality issues in software design. Scalability is defined as the web application's capacity to sustain the number of concurrent users and/or transactions, while sustaining sufficient response times to its users. To test scalability, web traffic loads must be determined in order to obtain the threshold requirement for scalability. Sometimes, existing traffic levels are used to simulate the load.

Availability Availability not only means that the web software is available at any time, but in today's environment it also must be available when accessed by diverse browsers. To be available in this sense, websites must adapt their presentations to work with all browsers, which require more knowledge and effort on the part of the developers.

Maintainability Web-based software has a much faster update rate. Maintenance updates can be installed and immediately made available to customers through the website. Thus, even small individual changes (such as changing the label on a button) can be installed immediately. One result of this is that instead of maintenance cycles of months or years, websites have maintenance cycles of days or even hours.

17.6 Web Engineering (WebE)

The diversity, complexity, and unique aspects of web-based software were not recognized and given due consideration by the developers early in the development of these systems. These systems also lack proper testing and evaluation. As a result, they faced a web crisis or what is called a *software crisis.*

> *Many organizations are heading toward a web crisis in which they are unable to keep the system updated and/or grow their system at the rate that is needed. This crisis involves the proliferation of quickly 'hacked together' web systems that are kept running via continual stream of patches or upgrades developed without systematic approaches.* [148]

Thus, to tackle the crisis in web-based system development, web engineering [150-156] as a new discipline has been evolved. This new discipline adopts a systematic approach to development of high-quality web-based systems.

> *Web engineering is an application of scientific, engineering, and management principles and disciplined and systematic approaches to the successful development, deployment, and maintenance of high-quality web-based systems and applications.* [149]

Web engineering activities involve all the aspects starting from the conception and development to the implementation, performance evaluation, and continual maintenance. We will discuss web-based analysis and design and then the testing of web-based systems in the subsequent sections.

17.6.1 Analysis and Design of Web-based Systems

Various methods have been proposed in the recent years for modeling web-based systems for analysis and design. Keeping in consideration the nature of web applications, the design of these systems must consider the following:

- Design for usability–interface design, navigation.
- Design for comprehension.
- Design for performance–responsiveness.
- Design for security and integrity.
- Design for evolution, growth, and maintainability.
- Design for testability.
- Graphics and multimedia design.
- Web page development.

It has been found that UML-based modeling [157-160] is more suitable according to the dynamic and complex nature of these systems. Therefore, we briefly mention all the UML-based models for analysis and design as discussed by Koch and Kraus [161]. The purpose to discuss them here is to get familiar with the analysis and design models in the context of web-based systems. Moreover, the understanding of these models will also help in testing these systems. As we have seen in testing for object-oriented software, testing of OO software was based on the models developed in analysis and design. On the same pattern, we will first understand UML-based analysis and design models and then move to testing these systems.

Conceptual Modeling

The conceptual model is a static view of the system that shows a collection of static elements of the problem domain. To model this, class and association modeling elements of UML are used. Additionally, role names, multiplicities, different forms of associations supported by the UML like aggregation, inheritance, composition, and association class can also be used to improve the modeling. If there are many classes in conceptual modeling, then packages may also be used. Since the conceptual model will show the objects involved in typical activities, use-cases with activity diagrams are also used for conceptual modeling.

Navigation Modeling

There are two types of navigation modeling: *navigation space model* and *navigation structure model*. The navigation space model specifies which objects can be visited by navigation through the application. It is a model at the analysis level. The navigation structure model defines how these objects are reached. It is a model at the design level.

The navigation models are represented by stereotyped class diagrams consisting of classes of those objects which can be visited by navigation through the web application and the associations which specify which objects can be

reached through navigation. The main modeling elements are the *stereotyped class «navigation class»* and *the stereotyped association «direct navigability»*. However, navigation model considers only those classes of the conceptual model that are relevant for navigation. Further, the navigation space model can be enhanced.

Presentation Modeling

This model transforms the navigation structure model in a set of models that show the static location of the objects visible to the user, i.e. a schematic representation of these objects. Using this presentation model, the discussion regarding interface design can be discussed with the customer. The sketches of these representations are drawn using a class diagram with UML composition notation for classes, i.e. containment represented by graphical nesting of the symbols of the parts within the symbol of the composite.

Web Scenarios Modeling

The dynamic behaviour of any web scenarios including the navigation scenarios are modeled here. The scenarios help in introducing more specification details. The navigation structure model is detailed here showing all the sequences of the behaviour of the objects, i.e. stimuli that trigger the transitions and explicitly including the actions to be performed. This modeling is done with the help of sequence interaction diagram/collaboration diagrams and state chart diagrams.

Task Modeling

The tasks performed by the user or system are modeled here. Basically, the use-cases as tasks are refined here. Since use-cases are refined by activity diagrams in UML, these diagrams are used for task modeling. The modeling elements for task modeling are those for activity diagrams, i.e. *activities, transitions, branche*s, etc. A task hierarchy can also be represented from coarse to fine-grained activities. The temporal order (with branches) between tasks is expressed by transitions between activities.

Task modeling also comprises of the description of the objects–called *referents*–that the user will perceive. For this purpose, presentation and conceptual objects are included in the task model. The relationship between tasks and these objects is expressed as object flow visualized as directed dashed lines. The ingoing presentation objects express user input through these presentation objects and outgoing presentation objects express output to the user through these presentation objects. The conceptual objects express input and output of tasks.

Configuration Modeling

Since web systems are diverse and complex in nature, we need to consider all the configuration and attributes that may be present on the client as well as on the server side. Moreover, some complex issues like deployment of process among multiple servers, compatibility with server OS, security issues, etc. are also considered. Deployment diagrams are used for configuration modeling.

17.6.2 Design Activities

Based on the discussion above, the following design activities have been identified for a web-based system [116]:

Interface Design

It considers the design of all interfaces recognized in the system including screen layouts. It also includes the types of navigation possible through the web system. All the interface design and navigational design issues are first discussed with the end-user. Interfaces should be designed such that they are flexible, consistent, and readable.

Content Design

All the contents regarding the web application identified as static elements in the conceptual modeling are designed here. The objects identified are known as *content objects*. Thus, these content objects and their relationships are designed here.

Architecture Design

The overall architecture of web applications is divided into the following three categories:

(i) An overall system architecture describing how the network and the various servers (web servers, application servers, and database servers) interact.

(ii) Content architecture design is needed to create a structure to organize the contents. Thus, this design is for structured presentation of the content objects. The content structure depends on factors like nature of the application, nature of the information, etc.

(iii) Application architecture is designed within the context of the development environment in which the web application is developed. This design provides an infrastructure so that functioning of the application in the form of components is clearly understood.

Presentation Design

This design is related to the look and feel of the application. The usability issues are considered here. The idea is that web application should be easy and flexible to use by its user. The design issues are related to layout of the web pages, graphics to be inserted, colour schemes, typefaces, etc.

Navigation design

Navigational paths are designed such that users are able to navigate from one place to another in the web application. The identified content objects, which are participating in navigation, become the navigation nodes and links between them define the navigation paths.

17.7 Testing of Web-Based Systems

Web-based systems have a different nature as compared to traditional systems. We have seen the key differences between them. Due to the environment difference and challenges of dynamic behaviour, complexity and diversity also makes the testing of these systems a challenge. These systems need to be tested not only to check whether it does what it is designed to do but also to evaluate how well it appears on the (different) web browsers. Moreover, they need to be tested for various quality parameters which are a must for these systems like security, usability, etc. Hence, a lot of effort is required for test planning and test designing.

Test cases should be written covering the different scenarios not only of the functional usage but also the technical implementation environment conditions such as network speeds, screen resolution, etc. For example, an application may work fine on Broadband Internet users but may perform miserably for users with dial-up connections. Web applications are known to give errors on slow networks, whereas they perform well on high-speed connections. Web pages don't render correctly for certain situations but work fine with others. Images may take longer to download for slower networks and the end-user perception of the application may not be good.

There may be a number of navigation paths possible through a web application. Therefore, all these paths must be tested. Along with multiple navigation paths, users may have varying backgrounds and skills. Testing should be performed keeping in view all the possible categories of users in view. This becomes the issues in usability testing.

Another issue of great concern is the security testing of web applications. There are two cases. For Intranet-based applications, there are no such threats on the application. However, in case of Internet-based applications, the users

may need to be authenticated and security measures may have to be much more stringent. Test cases need to be designed to test the various scenarios and risks involved.

Traditional software must be tested on different platforms, or it may fail in some platforms. Similarly, web-based software must be tested with all the dimensions which are making it diverse. For example, users may have different browsers while accessing the applications. This aspect also needs to be tested under compatibility testing. If we test the application only on Internet Explorer, then we cannot ensure that if works well on Netscape or other browsers. Because these browsers may not only render pages differently but also have varying levels of support for client side scripting languages such as Java Script.

The strategy for testing web-based systems is the same as for other systems, i.e. verification and validation. Verification largely considers the checking of analysis and design models which have been described above. Various types of testing derive from the design models only. However, the quality parameters are also important factors which form the other types of testing. We discuss these various types of web testing in the following sections.

17.7.1 Interface Testing

Interface is a major requirement in any web application. More importantly, the user interface with web application must be proper and flexible. Therefore as a part of verification, present model and web scenarios model must be checked to ensure all interfaces. The interfaces between the concerned client and servers should also be considered. There are two main interfaces on the server side: *web server and application server interface* and *application server and database server interface.*

Web applications establish links between the web server and the application server at the onset. The application server in turn connects to the database server for data retrieval, processing, and storage. It is an important factor that these connections or interfaces work seamlessly without any failure or degradation in performance of speed and accuracy. Testing should check for appropriate error messages, roll back in case of failure to execute or user interruption. The complexity of this test is in ensuring that the respective interface, be it web server or application or database interface, captures the errors and initiates the appropriate error messages to the web application.

Thus, in interface testing, all interfaces are checked such that all the interactions between these servers are executed properly. Errors are handled properly. If database or web server returns any error message for any query by the application server, then the application server should catch and display these

error messages appropriately to users. Check what happens if the user interrupts any transaction in between? Check what happens if connection to web server is reset in between? Compatibility of server with software, hardware, network, and database should also be tested.

17.7.2 Usability Testing

The presentation design emphasizing the interface between user and web application gives rise to usability testing. The actual user of application should feel good while using the application and understand every thing visible to him on it. Usability testing is not a functionality testing, but the web application is reviewed and tested from a user's viewpoint. The importance of usability testing can be realized with the fact that we can even lose users because of a poor design. For example, check that form controls, such as boxes and buttons, are easy to use, appropriate to the task, and provide easy navigation for the user. The critical point for designers and testers in this attesting is that web application must be as pleasant and flexible as possible to the user.

Usability testing may include tests for navigation. It refers to how the user navigates the web pages and uses the links to move to different pages. Besides this, content should be logical and easy to understand. Check for spelling errors. Use of dark colours annoy users and should not be used in the site theme. You can follow some standards that are used for web page and content building. Content should be meaningful. All the anchor text links should work properly. Images should be placed properly with the proper sizes.

For verification of web application, the presentation design must be checked properly so that most of the errors are resolved at the earlier stages only. Verification can be done with a technique called *card-sorting technique* given by Michael D. Levi and Frederick G. Conrad. According to this technique, a group of end-users are given a set of randomly ordered index cards, each of which is labeled with a concept from the task domain. The users scatter all the index cards on the desk, sort them according to a category. Further arrange these groups in broader category. Write a name for each of the larger groupings, write it on a slip of paper, and attach each slip to the corresponding group. After this process of sorting, compare the card sort results to the original presentation design of the application. In this comparison, we may find several areas where we can improve the underlying hierarchy so that users can easily find the information they were looking for.

For validation, a scenario-based usability testing can be performed. This type of testing may take the help of use-cases designed in the use-case model for the system. All the use-cases covering usability points can become the base for designing test cases for usability testing. In this usability testing, some categories of users are invited to perform the testing. The testers meet the group

of participants to describe the system in general terms, give an overview of the process, and answer any questions. Participants are seated in front of a desktop computer and asked to work through the scenario questions. At the end of the session, a group discussion is held to note down the participants' reactions and suggestions for improvement. The results of user testing can also be taken from the participants in the form of a questionnaire. As they use the application, they answer these questions and give feedback to the testers in the end. Besides this, web server logs can also be maintained for the session of usage by the participants. It provides the testers with a time-stamped record of each participant's sessions providing an excellent approximation of users' journeys through a site.

The log mining method discussed above can be used for improvement in application even after the release of the product. The actual user session logs can be recorded and evaluated from the usability point. The general guidelines for usability testing are:

1. Present information in a natural and logical order.
2. Indicate similar concepts through identical terminology and graphics. Adhere to uniform conventions for layout, formatting, typefaces, labeling, etc.
3. Do not force users to remember key information across documents.
4. Keep in consideration that users may be from diverse categories with various goals. Provide understandable instructions where useful. Lay out screens in such a manner that frequently accessed information is easily found.
5. The user should not get irritated while navigating through the web application. Create visually pleasing displays. Eliminate information which is irrelevant or distracting.
6. Content writer should not mix the topics of information. There should be clarity in the information being displayed.
7. Organize information hierarchically, with more general information appearing before more specific detail. Encourage the user to delve as deeply as needed, but to stop whenever sufficient information has been received.
8. Check that the links are active such that there are no erroneous or misleading links.

17.7.3 Content Testing

The content we see on the web pages has a strong impression on its user. If these contents are not satisfactory to him, he may not visit the web page again.

Check the completeness and correctness properties of web application content. Check that certain information is available on a given web page, links between pages exist, or even check the existence of the web pages themselves (completeness property). Furthermore, web application content may need to be checked against semantic conditions to see if they meet the web document (correctness property). Therefore the contents should be correct, visible, flexible to use, organized, and consistent.

This type of testing targets the testing of static and dynamic contents of web application. Static contents can be checked as a part of verification. For instance, forms are an integral part of any website. Forms are used to get information from users and to keep interacting with them. First, check all the validations on each field. Check for the default values of fields and also wrong inputs to the fields in the forms. Options to create forms if any, form delete, view, or modify the forms must also be checked.

Static testing may consider checking the following points:

1. Various layouts.
2. Check forms for their field validation, error message for wrong input, optional and mandatory fields with specified length, buttons on the form, etc.
3. A table is present and has the expected number of rows and columns and pre-defined properties.
4. Grammatical mistakes in text description of web page.
5. Typographical mistakes.
6. Content organization.
7. Content consistency.
8. Data integrity and errors while you edit, delete, modify the forms.
9. Content accuracy and completeness.
10. Relationship between content objects.
11. Text contents.
12. Text fragments against formatting expectations. This differs slightly from simple text checking in that the formatting tags can be located loosely on the page as opposed to a fixed string for text content.
13. Graphics content with proper visibility.
14. Media contents to be placed at appropriate places.
15. All types of navigation links like internal links, external links, mail links, broken links to be placed at appropriate places.
16. All links on a web page are active.

A checklist for content verification can be prepared as was seen in Chapter 3.

There may also be dynamic contents on a web page. Largely, dynamic testing will be suitable in testing these dynamic contents. These dynamic contents can be in many forms. One possibility is that there are constantly changing contents, e.g. weather information web pages or online newspaper. Another case may be that web applications are generated dynamically from information contained in a database or in a cookie. Many web applications today work interactively in the manner that in response to a user request for some information, it interacts with some DBMS, extracts the relevant data, creates the dynamic content objects for this extracted data, and sends these content objects to the user for display. In the same manner, the information can be generated dynamically from cookies also, i.e. dynamic content objects for cookies are also there.

The problem in the design of these dynamic contents is that there may be many errors due to its dynamic behaviour. Therefore, testing of these dynamic contents becomes necessary to uncover the errors. Changing contents on a web page must be tested whether the contents are appearing every time in the same format. Moreover, there is consistency between the changed content and static content.

Test all database interface-related functionality for all dynamic content objects. Check if all the database queries are executing correctly, data is retrieved correctly, and also updated correctly. Load testing or performance testing can also be done on database.

Cookies are small files stored on the user machine. These are basically used to maintain the session, mainly the login sessions. The testing of the entire interface with these cookies must also be tested. Test the application by enabling or disabling the cookies in browser options. Test if the cookies are encrypted before writing to user machine. Check the effect on application security by deleting the cookies.

17.7.4 Navigation Testing

We have checked the navigation contents in interface testing. But to ensure the functioning of correct sequence of those navigations, navigation testing is performed on various possible paths in the web application. Design the test cases such that the following navigations are correctly executing:

1. Internal links
2. External links
3. Redirected links
4. Navigation for searching inside the web application

The errors must be checked during navigation testing for the following:

1. The links should not be broken due to any reason.
2. The redirected links should be with proper messages displayed to the user.
3. Check that all possible navigation paths are active.
4. Check that all possible navigation paths are relevant.
5. Check the navigations for the back and forward buttons, whether they are working properly.

17.7.5 Configuration/Compatibility Testing

Diversity in configuration for web applications makes the testing of these systems very difficult. As discussed above, there may be various types of browsers supporting different operating systems, variations in servers, networks, etc. Therefore, configuration testing becomes important so that there is compatibility between various available resources and application software. The testers must consider these configurations and compatibility issues so that they can design the test cases incorporating all the configurations. Some points to be careful about while testing configuration are:

1. There are a number of different browsers and browser options. The web application has to be designed to be compatible for majority of the browsers.
2. The graphics and other objects on a website have to be tested on multiple browsers. If more than one browser will be supported, then the graphics have to be visually checked for differences in the physical appearance. Some of the things to check are centering of objects, table layouts, colours, monitor resolution, forms, and buttons.
3. The code that executes from the browser also has to be tested. There are different versions of HTML. They are similar in some ways but they have different tags which may produce different features. Some of the other codes to be tested are Java, JavaScript, ActiveX, VBscripts, Cgi-Bin Scripts, and Database access. Cgi-Bin Scripts have to be checked for end-to-end operations and is most essential for e-commerce sites. The same goes for database access.
4. All new technologies used in the web development like graphics designs, interface calls like different API's, may not be available in all the operating systems. Test your web application on different operating systems like Windows, Unix, MAC, Linux, Solaris with different OS flavors.

17.7.6 Security Testing

Today, the web applications store more vital data and the number of transactions on the web has increased tremendously with the increasing number of users. Therefore, in the Internet environment, the most challenging issue is to protect the web applications from hackers, crackers, spoofers, virus launchers, etc. Through security testing, we try to ensure that data on the web applications remain confidential, i.e. there is no unauthorized access. Security testing also ensures that users can perform only those tasks that they are authorized to perform.

In a web application, the risk of attack is multifold, i.e. it can be on the web software, client-side environment, network communications, and server-side environments. Therefore, web application security is particularly important because these are generally accessible to more users than the desktop applications. Often, they are accessed from different locations, using different systems and browsers, exposing them to different security issues, especially external attacks. It means that web applications must be designed and developed such that they are able to nullify any attack from outside. Therefore, this issue is also related to testing of web application in terms of security. We need to design the test cases such that the application passes the security test.

Security Test Plan

Security testing can be divided into two categories: testing the security of the infrastructure hosting the web application and testing for vulnerabilities of the web application. Firewalls and port scans can be the solution for security of infrastructure. For vulnerabilities, user authentication, restricted and encrypted use of cookies, data communicated must be planned. Moreover, users should not be able to browse through the directories in the server.

Planning for security testing can be done with the help of some threat models. These models may be prepared at the time of requirement gathering and test plan can also be prepared correspondingly. These threat models will help in incorporating security issues in designing and later can also help in security testing.

Find all the component interfaces for performing security testing on a component. This is because most of the security bugs can be found on the interfaces only. The interfaces are then prioritized according to their level of vulnerability. The high-priority interfaces are tested thoroughly by injecting mutated data to be accessed by that interface in order to check the security.

While performing security testing, the testers should take care that they do not modify the configuration of the application or the server, services running on the server, and existing user or customer data hosted by the application.

Various Threat Types and their corresponding Test cases

Unauthorized user/Fake identity/Password cracking When an unauthorized user tries to access the software by using fake identity, then security testing should be done such that any unauthorized user is not able to see the contents/ data in the software.

Cross-site scripting (XSS) When a user inserts HTML/client-side script in the user interface of a web application and this insertion is visible to other users, it is called *cross-site scripting* (XSS). Attacker can use this method to execute malicious script or URL on the victim's browser. Using cross-site scripting, attacker can use scripts like JavaScript to steal user cookies and information stored in the cookies. To avoid this, the tester should additionally check the web application for XSS.

Buffer over flows Buffer overflow is another big problem when handling memory allocation if there is no overflow check on the client software. Due to this problem, malicious code can be executed by the hackers. In the application, check the buffer overflow module and the different ways of submitting a range of lengths to the application.

URL manipulation There may be chances that communication through HTTP is also not safe. The web application uses the HTTP GET method to pass information between the client and the server. The information is passed in parameters in the query string. Again, the attacker may change some information in query string passed from GET request so that he may get some information or corrupt the data. When one attempts to modify the data, it is known as fiddling of data. The tester should check if the application passes important information in the query string and design the test cases correspondingly. Write the test cases such that a general user tries to modify the private information.

SQL injection Hackers can also put some SQL statements through the web application user interface into some queries meant for querying the database. In this way, he can get vital information from the server database. Even if the attacker is successful in crashing the application, from the SQL query error shown on the browser, the attacker can get the information they are looking for. Design the test cases such that special characters from user inputs should be handled/escaped properly in such cases.

Denial of service When a service does not respond, it is denial of service. There are several ways that can make an application fail. For example, heavy load put on the application, distorted data that may crash the application, overloading of memory, etc. Design the test cases considering all these factors.

17.7.7 Performance Testing

Web applications' performance is also a big issue in today's busy Internet environment. The user wants to retrieve the information as soon as possible without any delay. This assumes the high importance of performance testing of web applications. Is application able to respond in a timely manner or is it ready to take the maximum load or beyond that? These questions imply that web applications must also be tested for performance. Performance testing helps the developer to identify the bottlenecks in the system and can be rectified.

In performance testing, we evaluate metrics like response time, throughput, and resource utilization against desired values. Using the results of this evaluation, we are able to predict whether the software is in a condition to be released or requires improvement before it is released. Moreover, we can also find the bottlenecks in the web application. Bottlenecks for web applications can be code, database, network, peripheral devices, etc.

Performance Parameters

Performance parameters, against which the testing can be performed, are given here:

Resource utilization The percentage of time a resource (CPU, Memory, I/O, Peripheral, Network) is busy.

Throughput The number of event responses that have been completed over a given interval of time.

Response time The time lapsed between a request and its reply.

Database load The number of times database is accessed by web application over a given interval of time.

Scalability The ability of an application to handle additional workload, without adversely affecting performance, by adding resources such as processor, memory, and storage capacity.

Round-trip time How long does the entire user-requested transaction take, including connection and processing time?

Types of Performance Testing

Performance tests are broadly divided into following categories:

Load testing Can the system sustain at times of peak load? The site should handle many simultaneous user requests, large input data from users, simultaneous connection to DB, heavy load on specific pages, etc. When

we need to test the application with these types of loads, then load testing is performed on the system. It focuses on determining or validating performance characteristics of the system when subjected to workloads and load volumes anticipated during production operations. It refers to how much load (the best example of load in web application is how many concurrent users) you can put on the web application and it will still serve flawlessly.

There are a few types of load testing which we can be performed. We can perform *capacity testing* to determine the maximum load the web service can handle before failing. Capacity testing reveals the web services' ultimate limit. We may also perform *scalability testing* to determine how effectively the web service will expand to accommodate an increasing load.

Stress testing Generally, stress refers to stretching the system beyond its specification limits. Web stress testing is performed to break the site by giving stress and check how the system reacts to stress and how the system recovers from crashes. It focuses on determining or validating performance characteristics of the system when subjected to conditions beyond those anticipated during production operations. It also tests the performance of the system under stressful conditions, such as limited memory, insufficient disk space, or server failure. These tests are designed to determine under what conditions an application will fail, how it will fail, and how gracefully it may recover from the failure. Some examples of graceful failure are: the system saves state at the time of failure and does not crash suddenly; On restarting it, the system recovers from the last good state; the system shows meaningful error messages to the user.

SUMMARY

Web-based systems are specialized software where conventional testing is not applicable. We need to understand the issues related to web-based system and applications in order to understand the testing issues and perform web-based testing. Web applications in today's environment are highly accessible by every type of users. Their uses demand available, reliable, and secure web applications. Due to these considerations, testing of web software becomes a challenge for the testers.

Web applications interact with many components that run on diverse hardware and software platforms. It is difficult to determine statically the application's control flow because the control flow is highly dependent on user input and sometimes in terms of trends on user behaviour over time or user location. Web applications often are affected by factors that may cause incompatibility and interoperability issues. The problem of incompatibility may exist on both the client as well as the server side. Thus, web applications are diverse, complex, dynamic, and face the problems of incompatibility and interoperability.

Web engineering is the solution for web-based systems like software engineering is for general software. Web engineering is the systematic discipline which includes the process of

analysing, designing, building, and testing the web-based systems. In this chapter, we have discussed the analysis and design models for the understanding of testing of web-based systems. After this, web testing types have been categorized taking analysis and design models as the base.

Let us review the important concepts described in this chapter:

- The web-based software system consists of a set of web pages and components that interact to form a system which executes using web server(s), network, HTTP, and a browser, and in which user input affects the state of the system.
- Web applications interact with many components that run on diverse hardware and software platforms.
- The key aspect of web applications is its dynamic nature. The dynamic aspects are caused by uncertainty in the program behaviour, changes in application requirements, rapidly evolving web technology itself, and other factors.
- Clients of web-based systems impose very short development time, compared to other software or information systems projects.
- Testing of web-based systems needs techniques considering the different nature of web software.
- Web engineering is the application of scientific, engineering, and management principles and disciplined and systematic approaches to the successful development, deployment, and maintenance of high-quality web-based systems and applications.
- Conceptual model is a static view of the system that shows a collection of static elements of the problem domain.
- The navigation space model specifies which objects can be visited by navigation through the application. It is a model at the analysis level.
- Presentation modeling transforms the navigation structure model into a set of models that show the static location of the objects visible to the user, i.e. a schematic representation of these objects.
- Web scenarios modeling models the dynamic behaviour of any web scenarios including the navigation scenarios.
- Task modeling models the tasks performed by the user or system. Basically, the use-cases as tasks are refined here.
- Configuration modeling considers the entire configuration and attributes that may be present on the client as well as the server side.
- Interface design considers the design of all interfaces recognized in the system including screen layouts.
- Content design considers the design of all the contents regarding the web application identified as static elements in the conceptual modeling.
- Architecture design considers the design of the overall system architecture of the web application, content architecture design needed to create a structure to organize the contents, and application architecture designed within the context of the development environment in which the web application is developed.

- Presentation design considers the design related to the look and feel of the application.
- Navigation design considers the design of navigational paths such that users are able to navigate from one place to another in the web application.
- Interface testing tests the user interface with web application, the interfaces between the concerned client and servers, server and application server interface, and application server and database server interface.
- Usability testing ensures that web application is as pleasant and flexible as possible to the user.
- Content testing checks that certain information is available on a given web page, links between pages exist, and also the existence of the web pages themselves (completeness property). Furthermore, web application content may need to be checked against semantic conditions to see if they are met by the web document (correctness property).
- Navigation testing ensures the functioning of correct sequence of navigations like internal links, external links, redirected links, navigation for searching inside the web application, etc.
- Configuration/compatibility testing checks the compatibility between various available resources and application software. The testers must consider these configurations and compatibility issues so that they can design the test cases incorporating all the configurations.
- Security testing tests that data on the web applications remain confidential. It also ensures that users can perform only those tasks that they are authorized to perform.
- Performance testing helps the developer to identify the bottlenecks in the system that can be rectified.
- Load testing focuses on determining or validating performance characteristics of the system when subjected to workloads and load volumes anticipated during production operations.
- Stress testing is performed to break the site by giving stress and check how the system reacts to the stress and how it recovers from crashes.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Web-based systems impose very _____ development time, compared to other software or information systems projects.
   (a) long
   (b) short
   (c) same
   (d) none of the above
2. The problem of incompatibility may exist on the
   (a) client side only

(b) server side only
(c) both client as well as server side
(d) none of the above

3. Conceptual model is a _____ view of the system that shows a collection of static elements of the problem domain.
   (a) static
   (b) dynamic
   (c) both
   (d) none of the above
4. The navigation space model is at the _____.
   (a) analysis level
   (b) design level
   (c) testing level
   (d) none of the above
5. Navigation structure model is at the _____.
   (a) analysis level
   (b) design level
   (c) testing level
   (d) none of the above
6. Presentation modeling transforms the _____ model in a set of models that show the static location of the objects visible to the user.
   (a) navigation structure
   (b) navigation space
   (c) object structure
   (d) none of the above
7. When one attempts to _____ the data, this is known as fiddling of data.
   (a) add
   (b) modify
   (c) delete
   (d) none of the above
8. Web applications require the presence of web server in simple configurations and multiple servers in more complex settings. Such applications are more precisely called _____.
   (a) web-enabled applications
   (b) website
   (c) web page
   (d) web-based applications

9. Applications, which may operate independent of any servers and rely on operating system services to perform their functions, are called _____.
   (a) web-enabled applications
   (b) website
   (c) web page
   (d) web-based applications
10. Simple client-server model was expanded first to _____ model and then the N-tier later on.
   (a) 3-tier
   (b) 4-tier
   (c) 5-tier
   (d) none of the above
11. A cookie is a _____ file on a user's system that identifies the user.
   (a) hyper-text
   (b) text
   (c) graphic
   (d) none of the above
12. _____ is defined as the web application's capacity to sustain the number of concurrent users and/or transactions, while sustaining sufficient response times to its users.
   (a) Availability
   (b) Maintainability
   (c) Scalability
   (d) none of the above
13. Web-based software has a much _____ update rate.
   (a) faster
   (b) slower
   (c) normal
   (d) none of the above
14. The navigation models are represented by stereotyped _____ diagrams.
   (a) activity
   (b) class
   (c) use-case
   (d) none of the above
15. Web-scenario modeling is done with the help of _____ diagrams.
   (a) interaction diagrams
   (b) state chart diagrams
   (c) activity diagrams
   (d) none of the above

Review Questions

1. What is the difference between web-based and web-enabled applications?
2. What is the difference among 2-tier, 3-tier, and N-tier web system architecture?
3. What is the difference between traditional software and web-based software?
4. 'Web-based systems are diverse, complex, and dynamic in nature'. Comment on this.
5. What are the quality aspects to be considered in web testing?
6. What is web crisis? Is it same as software crisis?
7. What is web engineering? Is it same as software engineering?
8. Take a web application project and prepare all the analysis and design models discussed in this chapter.
9. Consider all types of interfaces in the project taken in Question 8 and design the test cases for interface testing.
10. What will be the criteria for performing unit testing, integration testing, and system testing of a web application?
11. Design a checklist for verification of a web-based software.
12. List the quality aspects of a website and perform performance testing for it.
13. What are the possible types of vulnerabilities in a web-based system? How do we handle them in security testing?
14. What is the difference between load and stress testing?
15. Search material on testing tools used for various types of web testing. Prepare a comparison list of all of them.
16. What is the difference among XSS, buffer overflow, URL manipulation, and SQL injection?

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Part 7 Tracking the Bug

CHAPTER

Chapter 18:
Debugging

The testing process ends with the successful finding of bugs. But this process is a waste if we do not track this bug to find the exact location of errors. It means that the next immediate process after testing is debugging, wherein the bug is tracked to its exact location.

However, debugging as a process has not been given much attention. Due to this, debugging becomes a complex process. It is also said that debugging is more an art than a technique. It may be possible that debugging as an art may give better results sometimes, but not always. Moreover, the proficiency to use it as an art comes with experience. Thus, the time has come that we formalize the debugging methods as techniques, rather than randomized tracking of bug. This part discusses this issue and several debugging techniques.

This part will make ground for the following concepts:

- Debugging as an art or technique
- Debugging is a process
- Debugging techniques
- Debuggers

Chapter

18

Debugging

OBJECTIVES

After reading this chapter, you should be able to understand:

- Debugging is not testing and testing is not debugging
- Debugging is more an art than a technique
- Steps of a general debugging process
- Debugging with Memory Dump
- Debugging with Watch Points
- Backtracking method
- Debugging guidelines
- Various types of Debuggers

Debugging is not a part of the testing domain. Therefore, debugging is not testing. It is a separate process performed as a consequence of testing. But the testing process is considered a waste if debugging is not performed after the testing. Testing phase in the SDLC aims to find more and more bugs, remove the errors, and build confidence in the software quality. In this sense, testing phase can be divided into two parts:

1. Preparation of test cases, executing them, and observing the output. This is known as *testing*.
2. If output of testing is not successful, then a failure has occurred. Now the goal is to find the bugs that caused the failure and remove the errors present in the software. This is called *debugging*.

Debugging is the process of identification of the symptoms of failures, tracing the bug, locating the errors that caused the bug, and correcting these errors. Describing it in more concrete terms, debugging is a two-part process. It begins with some indication of the existence of an error. It is the activity of [2]:

1. Determining the exact nature of the bug and location of the suspected error within the program.
2. Fixing or repairing the error.

18.1 Debugging: An Art or Technique?

Since software testing has not reached a maturity-level discipline, debugging is also an unplanned activity in most of the projects. Due to the lack of attention given to the debugging process, it has been developed as an art more

than a technique. However, viewing the debugging process as an art has given better results as compared to technique. The art of analysing the bug history can help a lot in debugging. Similar projects, more often than not, have a history of being affected by similar bugs in the past. Moreover, thinking critically in testing and debugging also supports debugging as an art. We start with the symptoms of failures and critically examine every symptom that may lead to the location of the actual error.

18.2 Debugging Process

As discussed, the goal of debugging process is to determine the exact nature of failure with the help of symptoms identified, locate the bugs and errors, and finally correct it. The debugging process (see Fig. 18.1) is explained in the following steps:

Figure 18.1 Debugging process

- Check the result of the output produced by executing test cases prepared in the testing process. If the actual output matches with the expected output, it means that the results are successful. Otherwise, there is failure in the output which needs to be analysed.
- Debugging is performed for the analysis of failure that has occurred, where we identify the cause of the problem and correct it. It may be possible that symptoms associated with the present failure are not suf-

ficient to find the bug. Therefore, some additional testing is required so that we may get more clues to analyse the causes of failures.

- If symptoms are sufficient to provide clues about the bug, then the cause of failure is identified. The bug is traced to find the actual location of the error.
- Once we find the actual location of the error, the bug is removed with corrections.
- Regression testing is performed as bug has been removed with corrections in the software. Thus, to validate the corrections, regression testing is necessary after every modification.

18.3 Debugging is Difficult

Debugging is a time-consuming process and sometimes becomes frustrating if you are not able to find the cause of the failure. Here are some reasons why:

- Debugging is performed under a tremendous amount of pressure which may cause problems instead of leading to clues of the problem. This pressure happens due to the following reasons:
  - (i) Self-induced pressure to fix the suspected bug as early as possible since it is related to the individual performance and ego.
  - (ii) Organizational time-bound pressure to fix the bugs is always there.
- The gap between the faults and failures is very large in case of debugging of software systems. Without examining the whole program, we cannot locate the error directly. That is why, debugging starts with observing the failures for finding clues of the problem. With the clues, we then trace the bug and error. Moreover, the symptoms are also not clear after observing the failures. This makes debugging a time-consuming process.
- The complex design of the software affects the debugging process. Highly coupled and low-cohesive module is difficult to debug.
- Experience matters in the debugging process. A new member of the team will find it difficult or cumbersome as compared to experienced members.
- Sometimes, failures don't happen. This does not mean that there is no bug. What happens is that we are unable to reproduce that failure always. This type of bug consumes a lot of time.

18.4 Debugging Techniques

18.4.1 Debugging with Memory Dump

In this technique, a printout of all registers and relevant memory locations is obtained and studied. The storage locations are in octal or hexadecimal format. The relevant data of the program is observed through these memory locations and registers for any bug in the program. However, this method is inefficient. This method should be used as the last option. Following are some drawbacks of this method [2]:

1. There is difficulty of establishing the correspondence between storage locations and the variables in one's source program.
2. The massive amount of data with which one is faced, most of which is irrelevant.
3. It is limited to static state of the program as it shows the state of the program at only one instant of time.

18.4.2 Debugging with Watch Points

The program's final output failure may not give sufficient clues about the bug. It is possible that intermediate execution, at some points in the program, may provide sufficient cause of the problem. At a particular point of execution in the program, value of variables or other actions can be verified. These particular points of execution are known as *watch points*. Debugging with watch points can be implemented with the following methods:

Output statements In this method, output statements can be used to check the state of a condition or a variable at some watch point in the program. Therefore, output statements are inserted at various watch points; program is complied and executed with these output statements. Execution of output statements may give some clues to find the bug. This method displays the dynamics of a program and allows one to examine information related to the program failure. This method has the following drawbacks:

(i) It may require many changes in the code. These changes may mask an error or introduce new errors in the program.

(ii) After analysing the bug, we may forget to remove these added statements which may cause other failures or misinterpretations in the result.

Breakpoint execution This is the advanced form of the watch point used with an automated debugger program. *Breakpoint* is actually a watch point inserted at various places in the program. But these insertions are not placed in the actual user program and therefore need not be removed manually like output statements.

The other difference in this method is that program is executed up to the breakpoint inserted. At that point, you can examine whatever is desired. Afterwards, the program will resume and will be executed further for the next breakpoint. Thus, breakpoints allow the programmer to control execution of the program and specify how and where the application will stop to allow further examination. Since there can be many breakpoints in the same program, all breakpoints can also be viewed by the programmer. Breakpoints displayed in the view show the source location of each breakpoint as well as its status.

Breakpoints have an obvious advantage over output statements. Some are discussed here:

(a) There is no need to compile the program after inserting breakpoints, while this is necessary after inserting output statements.

(b) Removing the breakpoints after their requirement is easy as compared to removing all inserted output statements in the program.

(c) The status of program variable or a particular condition can be seen after the execution of a breakpoint as the execution temporarily stops after the breakpoint. On the other hand in case of output statements, the full program is executed and output is viewed after the total execution of the program.

Breakpoints can be categorized as follows:

(i) *Unconditional breakpoint:* It is a simple breakpoint without any condition to be evaluated. It is simply inserted at a watch point and its execution stops the execution of the program.

(ii) *Conditional breakpoint:* On the activation of this breakpoint, one expression is evaluated for its Boolean value. If true, the breakpoint will cause a stop; otherwise, execution will continue.

(iii) *Temporary breakpoint:* This breakpoint is used only once in the program. When it is set, the program starts running, and once it stops, the temporary breakpoint is removed. The temporary nature is one of its attributes. In other respects, it is just like other breakpoints.

(iv) *Internal breakpoint:* These are invisible to the user but are key to debugger's correct handling of its algorithms. These are the breakpoints set by the debugger itself for its own purposes.

Single stepping The idea of single stepping is that the users should be able to watch the execution of the program after every executable instruction. After every instruction execution, the users can watch the condition or status of variable. Single stepping is implemented with the help of internal breakpoints.

Step-into It means execution proceeds into any function in the current source statement and stops at the first executable source line in that function.

Step-over It is also called skip, instead of step. It treats a call to a function as an atomic operation and proceeds past any function calls to the textually succeeding source line in the current scope.

18.4.3 BACKTRACKING

This is a logical approach for debugging a program. The following are the steps for backtracking process (see Fig. 18.2):

Figure 18.2 Backtracking

(a) Observe the symptom of the failure at the output side and reach the site where the symptom can be uncovered. For example, suppose you are not getting display of a value on the output. After examining the symptom of this problem, you uncover that one module is not sending the proper message to another module. It may also be the case that function *X* is not working. This can be uncovered using the above mentioned techniques, e.g. using breakpoints.

(b) Once you have reached the site where symptom has been uncovered, trace the source code starting backwards and move to the highest level of abstraction of design. The bug will be found in this path. For tracing backwards, collect source code of all the modules which are related at this point.

(c) Slowly isolate the module through logical backtracking using data flow diagrams (DFDs) of the modules wherein the bug resides.

(d) Logical backtracking in the isolated module will lead to the actual bug and error can thus be removed.

Backtracking requires that the person debugging must have knowledge regarding the design of the system so that he can understand the logical flow of the program through DFDs. This method also requires the manual observation of the source code. This method is very effective as compared to the

other methods in pinpointing the error location quickly, if you understand the logic of the program.

18.5 Correcting the Bugs

The second phase of the debugging process is to correct the error when it has been uncovered. But it is not as easy as it seems. The design of the software should not be affected by correcting the bug or due to new modifications. Before correcting the errors, we should concentrate on the following points:

(a) Evaluate the coupling of the logic and data structure where corrections are to be made. Highly coupled module correction can introduce many other bugs. That is why low-coupled module is easy to debug.

(b) After recognizing the influence of corrections on other modules or parts, plan the regression test cases to perform regression testing as discussed earlier.

(c) Perform regression testing with every correction in the software to ensure that the corrections have not introduced bugs in other parts of the software.

18.5.1 Debugging Guidelines

Fresh thinking leads to good debugging Don't spend continuous hours and hours on debugging a problem. The continuous involvement in the problem will collapse your efficiency. So rest in between and start with a fresh mind to concentrate on the problem.

Don't isolate the bug from your colleagues It has been observed that bugs cannot be solved in isolation. Show the problem to other colleagues, explain to them what is happening, and discuss the solution. There will be a lot of clues that have not been tried yet. It has also been observed that by just discussing the problem with others may suddenly lead you to come up with some solution. Therefore, it is always better to share the problem to others.

Don't attempt code modifications in the first attempt Debugging always starts with the analysis of clues. If you don't analyse the failures and clues and simply change the code randomly with the view–'let's see what happens with this change', this thinking will lead you nowhere but with junk code added to the software.

Additional test cases are a must if you don't get the symptom or clues to solve the problem Design test cases with the view to execute those parts of the program

which are causing the problem. This test case execution will provide symptoms to be analysed.

Regression testing is a must after debugging Don't simply fix the error and forget about it. The process of debugging does not end with fixing the error. After fixing the error, you need to realize the effects of this change on other parts. Test cases must be designed to uncover any effects on bugs produced on other parts. That is why regression testing is necessary after fixing the errors.

Design should be referred before fixing the error Any change in the code to fix the bug should be according to pre-specified design of the software. While making corrections, that design should not be violated by any means.

18.6 Debuggers

In the process of debugging, the help of debugging tools can also be taken to speed up the process. Debugger is a tool to help track down, isolate, and remove bugs from the software program. It controls the application being debugged so as to allow the programmer to follow the flow of program execution and at any desired point, stop the program and inspect the state of the program to verify its correctness. Debuggers can be used to

- Illustrate the dynamic nature of a program.
- Understand a program as well as to find and fix the bugs.
- Control the application using special facilities provided by the underlying operating system to give the use very fine control over the program under test.

18.6.1 Types of Debuggers

Kernel debugger It is for dealing with problems with an OS kernel on its own or for interactions between heavily OS-dependent application and the OS.

Basic machine-level debugger It is used for debugging the actual running code as they are processed by the CPU.

In-circuit emulator It emulates the system services so that all interactions between an application and the system can be monitored and traced. In-circuit emulators sit between the OS and the bare hardware and can watch and monitor all processes and all interactions between applications and OS.

Interpretive programming environment debugger Debugger is well integrated into the runtime interpreter and has very tight control over the running application.

SUMMARY

It is generally considered that debugging does not come under the domain of software testing. However, testing is wasteful if you do not remove the bugs reported and perform debugging. Therefore, debugging as a consequent process of testing is important to understand, both for the developers and the testers. This chapter shows the importance of the debugging process and its general process steps. Debugging has evolved with many techniques but it still gives better results if performed as an art. Various debugging techniques and debuggers have been discussed in this chapter along with the guidelines to perform the debugging.

Let us have a quick review of the important concepts described in this chapter:

- Debugging is the process of identification of symptoms of failures, tracing the bug, locating the errors that caused the bug, and correcting these errors.
- In debugging with the memory dump technique, printout of all registers and relevant memory locations is obtained and studied for any bug in the program.
- In debugging with watch points, at a particular point of execution in the program, value of variables or other actions can be verified. These particular points of execution are known as watch points.
- Output statements can be used to check the state of a condition or a variable at some watch point in the program.
- Breakpoint is actually a watch point inserted at various places in the program. But these insertions are not placed in the actual user program and therefore need not be removed manually like output statements.
- In single stepping, the users are able to watch execution of the program after every executable instruction.
- Debugger is a tool to help track down, isolate, and remove bugs from the software program.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Debugging is a part of testing domain.
   (a) true
   (b) false
   (c) none of the above
2. Viewing the debugging process as an art has given _____ results, as compared to technique.
   (a) poor
   (b) better
   (c) satisfactory
   (d) none of the above

3. The debugging process is to determine the exact nature of failure with the help of _____ identified.
   (a) symptoms
   (b) test cases
   (c) bugs
   (d) none of the above
4. Highly coupled and low-cohesive module is _____ to debug.
   (a) easy
   (b) difficult
   (c) cannot be debugged
   (d) none of the above
5. Technique in which a printout of all registers and relevant memory locations is obtained and studied, is called _____.
   (a) debugging
   (b) backtracking
   (c) debugging with watch points
   (d) debugging with memory dump
6. The technique in which the value of variables or other actions can be verified at a particular point of execution in the program is called _____.
   (a) debugging
   (b) backtracking
   (c) debugging with watch points
   (d) debugging with memory dump
7. Backtracking requires the debugger to have the knowledge of _____ of the system.
   (a) design
   (b) testing
   (c) code
   (d) none of the above

Review Questions

1. What is the importance of the debugging process?
2. Discuss the steps required for performing a debugging process.
3. Why is debugging considered a difficult process?
4. Take a debugger and give examples of all breakpoints in it, if supported.
5. Backtracking is considered an important debugging method. Take a project and list of its reported failures. Solve these failures with the help of backtracking.
6. Discuss various types of debuggers.

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

INCOME TAX CALCULATOR
A Case Study

Step

1

Introduction to Case Study

All the techniques learnt in this book can be practised using a case study. For this purpose, a case study of *Income Tax Calculator* application has been taken. The application has been designed and developed for the readers and all the test case design techniques have been applied on it. However, the application presented and implemented is only for illustrative purposes and it is not claimed that this application is free from defects and can be used practically for calculating the income tax of a person. The idea is only to present a working application and show how to perform testing on it.

The case study has been presented in the following sequence:

Requirement Specifications and Verification

The requirements for the case study have been collected and SRS ver 1.0 was prepared initially. The tax slabs and other details in this case study have been compiled from *www.incometaxindia.gov.in*. This draft of SRS was in a raw form. After this, verification on SRS ver 1.0 was performed and found that many features were not present in SRS. During verification on SRS, the checklist presented in Appendix has been used. The readers are advised that they should also perform verification using checklists and find some more deficiencies in SRS. In this way, SRS ver 2.0 was prepared as a result of verification on SRS ver 1.0. Another round of verification was performed on SRS ver 2.0 and finally we get SRS ver 3.0.

The readers are advised to prepare an SDD of this application and perform verification exercises on it to and get a final version of SDD.

Black-box Testing on SRS ver 3.0

Once the SRS is prepared, some black-box test cases have been designed using the techniques studied in Chapter 4. The test cases can be executed on the implemented executable application. The executable application can be directly taken from the CD.

Source Code

The application based on SRS ver 3.0 has been implemented in C language. There are two files: *TaxCalculator.c* and *Taxcalculator.h*. The readers can get these files directly from the CD and use and modify them the way they want.

White-Box Testing

The source code of *TaxCalculator.h* has been tested using white-box testing techniques. All the major white-box testing techniques have been applied on this source code. The test cases can be executed on the implemented executable application. The executable application along with the source code of application can be taken directly from the CD.

The readers should follow this sequence for studying the full case study and learn the testing techniques presented in this book. The case study provides the way to learn the testing concepts and techniques in a practical way.

Step

2

Income Tax Calculator SRS ver 1.0

A system is proposed to calculate the income tax of a person residing in India, provided his salary, savings, status, and donations are known. The system will accept personal details, income details, savings details and calculate total salary, net tax payable, educational cess, and hence the total tax payable.

Income Tax slabs 2009/2010 for Men

Income: up to 1.5 lacs	NO INCOME TAX
Income : 1.5 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Income Tax slabs 2009/2010 for Women

Income : up to 1.8 lacs	NO TAX
Income : 1.8 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Income Tax slabs 2009/2010 for Senior Citizen

Income : up to 2.25 lacs	NO TAX
Income : 2.25 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

In addition to the income tax calculated according to the above income tax slabs, a 3% of education cess will be charged on the total income tax paid (not on the total taxable income). If the taxable income exceeds Rs 10 lacs, a 10% surcharge on the total income tax (not on the total taxable income) is also charged.

Donations with 100% rebate

- The Prime Minister's National Relief Fund.
- The Prime Minister's Armenia Earthquake Relief Fund.
- The Africa (Public Contributions-India) Fund.
- The National Foundation for Communal Harmony.
- A university or any educational institution of national eminence as maybe approved by the prescribed authority. Please note that the prescribed authority in case of a university or a non-technical institution of national eminence is the Director-General (Income-Tac exemption) in concurrence with the Secretary, UGC. In case of any technical institution of national eminence, the prescribed authority is the Director-General (Income-Tax Exemption) in concurrence with the Secretary, All India Council of Technical Education.
- The Maharashtra Chief Minister's Earthquake Relief Fund.
- Any Zila Saksharta Samiti constituted in any district under the chairmanship of the Collector of that district for the purpose of improvement of primary education in villages and towns in such a district and for literacy and post literacy activities.
- The National Blood Transfusion Council or any State Blood Transfusion council whose sole objective is the control, supervision, regulation, or encouragement in India of the services related to operation and requirements of blood banks.
- Any fund set up by a State Government to provide medical relief to the poor.
- The Army Central Welfare Fund or the Indian Naval Benevolent Fund or the Air Force Central Welfare Fund established by the armed forces of the Union for the welfare of the past and present members of such forces or their dependants.
- The Andhra Pradesh Chief Minister's Cyclone Relief Fund, 1996.
- The National Illness Assistance Fund.
- The Chief Minister's Relief Fund or the Lieutenant Governor's Relief Fund in any State or Union Territory.
- The Government, or any local authority, institution or association as maybe approved by the Central Government for the purpose of promoting family planning.

Step

3

Verification on Income Tax Calculator SRS ver 1.0

Verification on Income Tax Calculator SRS ver 1.0 is presented here. The reader is advised to use checklists provided in the Appendix while performing verification and use his/her intelligence. The missing features found in this verification are highlighted.

A system is proposed to calculate the income tax of a person residing in India provided his salary, savings, status, and donations are known. The system will accept personal details, income details, savings details and calculate total salary, net tax payable, educational cess, and hence total tax payable.

Is the software meant only for salaried person or for anyone? There is no mention about the functional flow of the system about how it works. Who will interface with the system? How are the savings considered in calculating the net tax? There is no high-level functionality diagram representing interfaces and data flow.

Income Tax slabs 2009/2010 for Men

Income: up to 1.5 lacs	NO INCOME TAX
Income : 1.5 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Income Tax slabs 2009/2010 for Women

Income : up to 1.8 lacs	NO TAX
Income : 1.8 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Income Tax slabs 2009/2010 for Senior Citizen

Income : up to 2.25 lacs	NO TAX
Income : 2.25 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

In addition to the income tax calculated according to the above income tax slabs, a 3% of education cess will be charged on the total income tax paid (not on the total taxable income). If the taxable income exceeds Rs 10 lacs, a 10% surcharge on the total income tax (not on the total taxable income) is also charged

Donations with 100% rebate

- The Prime Minister's National Relief Fund.
- The Prime Minister's Armenia Earthquake Relief Fund.
- The Africa (Public Contributions-India) Fund.
- The National Foundation for Communal Harmony.
- A University or any educational institution of national eminence as maybe approved by the prescribed authority. In case of any technical institution of national eminence, the prescribed authority is the Director General (Income-Tax Exemption) in concurrence with the Secretary, All India Council of Technical Education.
- The Maharashtra Chief Minister's Earthquake Relief Fund.
- Any Zila Saksharta Samiti constituted in any district under the chairmanship of the Collector of that district for the purpose of improvement of primary education in villages and towns in such a district and for literacy and post literacy activities.
- The National Blood Transfusion Council or any State Blood Transfusion council whose sole objective is the control, supervision, regulation, or encouragement in India of the services related to operation and requirements of blood banks.
- Any fund set up by a State Government to provide medical relief to the poor.
- The Army Central Welfare Fund or the Indian Naval Benevolent Fund or the Air Force Central Welfare Fund established by the armed forces of the Union for the welfare of the past and present members of such forces or their dependants.
- The Andhra Pradesh Chief Minister's Cyclone Relief Fund, 1996.
- The National Illness Assistance Fund.
- The Chief Minister's Relief Fund or the Lieutenant Governor's Relief Fund in any State or Union Territory.
- The Government, or any local authority, institution or association as maybe approved by the Central Government for the purpose of promoting family planning.

What will be the rebate if the donation is not in the above list? In addition, the following items are missing in the SRS which are necessary to avoid bugs and misunderstanding:

1. High-level diagrams depicting external and internal interfaces.
2. The user interaction with the system.
3. Software Functions/Features.
4. Inputs and outputs formats and their ranges.
5. Software/Hardware Requirements.

Step

4

Income Tax Calculator SRS ver 2.0

A system is proposed to calculate the income tax of a person residing in India provided his income, savings, status, and donations are known. The system will accept personal details, income details, and savings details, and calculate the total salary, net tax payable, educational cess, and hence the total tax payable. The user gets the information about total tax to be paid.

The system will first accept personal details, income, donations, and savings. For donations, it provides a list of categories in which 100% rebate is provided. The user will look for the option provided and inform the system whether the donation lies in that list. If the donation lies in the list, 100% rebate will be provided, otherwise 50%. The system will check whether the savings are less than Rs 1 lac. If yes, then the whole amount will be deducted from the taxable income. Otherwise, Rs 1 lac will be deducted. Then the system will calculate the total tax and check if it exceeds Rs 10 lacs. If yes, a 10% surcharge on the total income tax (not on the total taxable income) is also charged and a 3% of education cess will be charged on the total income tax paid (not on the total taxable income). Finally, the system will show the net tax as per the following details:

Income Tax slabs 2009/2010 for Men

Income: up to 1.5 lacs	NO TAX
Income : 1.5 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Income Tax slabs 2009/2010 for Women

Income : up to 1.8 lacs	NO TAX
Income : 1.8 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Income Tax slabs 2009/2010 for Senior Citizen

Income : up to 2.25 lacs	NO TAX
Income : 2.25 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Donations with 100% rebate

- The Prime Minister's National Relief Fund.
- The Prime Minister's Armenia Earthquake Relief Fund.
- The Africa (Public Contributions-India) Fund.
- The National Foundation for Communal Harmony.
- A university or any educational institution of national eminence as maybe approved by the prescribed authority. In case of any technical institution of national eminence, the prescribed authority is the Director General (Income-Tax Exemption) in concurrence with the Secretary, All India Council of Technical Education.
- The Maharashtra Chief Minister's Earthquake Relief Fund.
- Any Zila Saksharta Samiti constituted in any district under the chairmanship of the Collector of that district for the purpose of improvement of primary education in villages and towns in such a district and for literacy and post literacy activities.
- The National Blood Transfusion Council or any State Blood Transfusion council whose sole objective is the control, supervision, regulation, or encouragement in India of the services related to operation and requirements of blood banks.
- Any fund set up by a State Government to provide medical relief to the poor.
- The Army Central Welfare Fund or the Indian Naval Benevolent Fund or the Air Force Central Welfare Fund established by the armed forces of the Union for the welfare of the past and present members of such forces or their dependants.

- The Andhra Pradesh Chief Minister's Cyclone Relief Fund, 1996.
- The National Illness Assistance Fund.
- The Chief Minister's Relief Fund or the Lieutenant Governor's Relief Fund in any State or Union Territory.
- The Government, or any local authority, institution or association as maybe approved by the Central Government for the purpose of promoting family planning.

Functional Requirements

Accept Personal Details

The function will accept the following details to be entered by user.

- Name (3 to 15 alphabets with spaces in between)
- Date of Birth (dd/mm/yyyy)
- Permanent address (3 to 30 characters)
- Sex (M/F one alphabet only)
- Status: Salaried or not (Y/N one alphabet only)

If the user enters the answer Y (Yes) to the status entry, then the function will display the following three entries, otherwise it will not.

- Designation (if Salaried) (3 to 15 alphabets)
- Name of the employer (if salaried) (3 to 25 alphabets with spaces)
- Address of the employer (if salaried) (3 to 30 characters)
- PAN number (10 characters including alphabets and digits 0–9)

- TDS circle where annual return/statement under section 206 is to be filed (3 to 15 alphabets with spaces)
- Period: From (dd/mm/yyyy)
  To (dd/mm/yyyy)
  Assessment year (yyyy-yy)

Accept Income Details

The function will enquire whether the user is a salaried person or has some other source of income. If the user is not a salaried person, the system will ask for the source of income. The user may enter various types of source of incomes as given below:

Source of Income: (3 to 20 alphabets with spaces)

Amount: (positive real numbers with maximum two decimal places)

The function will aggregate all the amounts of income as gross total income.

If the person is salaried, the function asks for the following details:

1. **Gross Salary**
   (a) Salary as per the provisions contained in the section 17(1)
   (b) Value of the prerequisites under section 17(2) (As per form number 12BA, wherever applicable)
   (c) Profits in lieu of salary under section 17(3) (As per form number 12BA, wherever applicable)
   (d) Total (to be calculated by this function)
2. **Less allowance to the extent exempt under section 10**
   This function will add the exempted allowances.
3. **Balance**
   This function will calculate the difference of the gross salary and the exempted allowances.
4. **Deductions**
   - Entertainment allowance
   - Tax on employment
5. **Aggregate**
   This function will calculate the aggregate of the deductions entered above.
6. **Income Chargeable Under The Head 'Salaries'**
   The function will calculate the difference of item 3–item 5.
7. **Add**
   Any other item reported by the employee.
   User may enter multiple incomes. The function will add all these incomes.
8. **Gross Total Income**
   The function will add item 6 and 7.
   All the amounts will be positive real numbers with maximum 2 decimal places.

Accept Savings & Donations Details

The function will ask the user to enter the total savings and the donations in the following format.

Saving type (3 to 20 alphabets with spaces)

Deductible amount (positive real numbers with maximum two decimal places)

The user may enter multiple savings. The function will add all the deductible amounts in aggregate deductible amount.

Accept Tax Deduction Details

If the person is salaried, then this function will accept the details if tax deducted by the employer during the year is in the following format:

Amount of tax deposited (positive real numbers with maximum two decimal places)

Date (dd/mm/yyyy)

Challan Number (5 to 20 characters)

The above details may be entered multiple times. The function will add all the amounts of tax deposited.

Amount of TDS (positive real numbers with maximum two decimal places)

The function will add all the amounts of tax deposited and the amount of TDS in the total tax deducted.

Generate Tax Payable Details

This function calculates the tax payable by the person in the following format:

Taxable income Function will calculate this by taking difference of gross total income in the function Accept Income Details and aggregate deductible amount in the function Accept Savings & Donation Details.

Tax on taxable income Function will calculate this using the appropriate slab of user as given above.

Surcharge Function will calculate the surcharge as if tax on taxable income exceeds Rs 10 lacs, a 10% surcharge is charged.

Education cess Function will calculate the education cess as a 3% of tax on taxable income.

Tax payable Function will sum up tax on taxable income, surcharge, and education cess.

Relief under section 89 User will enter the amount, if applicable.

Tax payable after relief (if applicable) Function will deduct relief amount from tax payable.

Total tax deducted Displayed from the function Accept Tax Deduction Details.

Tax payable/refundable The function will find the difference of tax payable and the total tax deducted. If the difference is positive, then this amount is the net tax to be paid by the person, otherwise the amount is due on the government to be refunded.

Step

5

Verification on Income Tax Calculator SRS ver 2.0

Verification on Income Tax Calculator SRS ver 2.0 is presented here. The reader is advised to use checklists while performing verification and use his/her intelligence. The missing features found in this verification are highlighted.

A system is proposed to calculate the income tax of a person residing in India provided his income, savings, status, and donations are known. The system will accept personal details, income details, savings details, and calculate total salary, net tax payable, educational cess, and hence total tax payable. The user gets the information about total tax to be paid.

The system will first accept personal details, income, donations, and savings. For donations, it provides a list of categories in which 100% rebate is provided. The user will look for the option provided and informs the system whether the donation lies in that list. If the donation lies in the list, 100% rebate will be provided otherwise 50%. The system will check whether the savings are less than Rs 1 lac. If yes, then whole amount will be deducted from the taxable income. Otherwise, Rs 1 lac will be deducted. Then the system will calculate the total tax and checks if it exceeds Rs 10 lacs, a 10% surcharge on the total income tax (not on the total taxable income) is also charged and a 3% of education cess will be charged on the total income tax paid (not on the total taxable income). Finally, the system will show the net tax as per the following details:

Income Tax slabs 2009/2010 for Men

Income: up to 1.5 lacs	NO TAX
Income : 1.5 lacs to 3 lacs	10 %
Income : 3 lacs to 5 lacs	20 %
Income : above 5 lacs	30 %

Income Tax slabs 2009/2010 for Women

Income : up to 1.8 lacs	NO TAX
Income : 1.8 lacs to 3 lacs	10 %
Income : 3 lacs to 5 lacs	20 %
Income : above 5 lacs	30 %

Income Tax slabs 2009/2010 for Senior Citizen

Income : up to 2.25 lacs	NO TAX
Income : 2.25 lacs to 3 lacs	10 %
Income : 3 lacs to 5 lacs	20 %
Income : above 5 lacs	30 %

Donations with 100% rebate

- The Prime Minister's National Relief Fund.
- The Prime Minister's Armenia Earthquake Relief Fund.
- The Africa (Public Contributions-India) Fund.
- The National Foundation for Communal Harmony.
- A University or any educational institution of national eminence as maybe approved by the prescribed authority. In case of any technical institution of national eminence, the prescribed authority is the Director General (Income-Tax Exemption) in concurrence with the Secretary, All India Council of Technical Education.
- The Maharashtra Chief Minister's Earthquake Relief Fund.
- Any Zila Saksharta Samiti constituted in any district under the chairmanship of the Collector of that district for the purpose of improvement of primary education in villages and towns in such a district and for literacy and post literacy activities.
- The National Blood Transfusion Council or any State Blood Transfusion council whose sole objective is the control, supervision, regulation, or encouragement in India of the services related to operation and requirements of blood banks.
- Any fund set up by a State Government to provide medical relief to the poor.

- The Army Central Welfare Fund or the Indian Naval Benevolent Fund or the Air Force Central Welfare Fund established by the armed forces of the Union for the welfare of the past and present members of such forces or their dependants.
- The Andhra Pradesh Chief Minister's Cyclone Relief Fund, 1996.
- The National Illness Assistance Fund.
- The Chief Minister's Relief Fund or the Lieutenant Governor's Relief Fund in any State or Union Territory.
- The Government, or any local authority, institution or association as maybe approved by the Central Government for the purpose of promoting family planning.

Functional Requirements

Accept Personal Details

The function will accept the following details to be entered by user.

- Name (3 to 15 alphabets with spaces in between)
- Date of Birth (dd/mm/yyyy)
- Permanent address (3 to 30 characters)

The address should not contain any character. The allowed characters like alphabets, digits, spaces, and commas should be mentioned.

- Sex (M/F one alphabet only)
- Salaried or not (Y/N one alphabet only)

- Designation (if Salaried) (3 to 15 alphabets)
- Name of the employer (if salaried) (3 to 25 alphabets with spaces)
- Address of the employer (if salaried) (3 to 30 characters)
- PAN number (10 characters including alphabets and digits 0-9)

Including means any character can be entered. The word should be *consisting.*

- TDS circle where annual return/ statement under section 206 is to be filed (3 to 15 alphabets with spaces)
- Period: From (dd/mm/yyyy)
- To (dd/mm/yyyy)
- Assessment year (yyyy-yy)

It is not clear whether the user can make a wrong entry and move ahead to the next entry or he cannot move ahead until he enters a correct entry.

Accept Income Details

The function will enquire whether the user is a salaried person or has some other source of income. If the user is not a salaried person, the system will ask for the source of income. The user may enter various types of source of incomes as given below:

Source of Income: (3 to 20 alphabets with spaces)

Amount: (positive real numbers with maximum two decimal places)

The function will aggregate all the amounts of income as gross total income. If the person is salaried, the function asks for the following details:

1. **Gross Salary**
   (a) Salary as per the provisions contained in the section 17(1)
   (b) Value of the prerequisites under section 17(2) (As per form number 12BA, wherever applicable)
   (c) Profits in lieu of salary under section 17(3) (As per form number 12BA, wherever applicable)
   (d) Total (to be calculated by this function)
2. **Less allowance to the extent exempt under section 10**

   This function will add the exempted allowances.
3. **Balance**

   This function will calculate the difference of the gross salary and the exempted allowances.
4. **Deductions**
   - Entertainment allowance
   - Tax on employment

5. **Aggregate**

   This function will calculate the aggregate of the deductions entered above.

6. **Income Chargeable Under the Head 'Salaries'**

   The function will calculate the difference of item 3 – item 5.

7. **ADD**

   Any other item reported by the employee

   User may enter multiple incomes. The function will add all these incomes.

8. **Gross Total Income**

   The function will add item 6 and 7.

   All the amounts will be positive decimal numbers with maximum 2 decimal places.

It is not clear whether the user can make a wrong entry and move ahead on the next entry or he cannot move ahead until he enters a correct entry.

Accept Savings & Donations Details

The function will ask the user to enter the total savings and the donations in the following format.

Saving Type: (3 to 20 alphabets with spaces)

Deductible amount: (positive real numbers with maximum two decimal places)

The user may enter multiple savings. The function will add all the deductible amounts in aggregate deductible amount.

It is not clear whether the user can make a wrong entry and move ahead to the next entry or he cannot move ahead until he enters a correct entry.

Accept Tax Deduction Details

If the person is salaried, then this function will accept the details if tax deducted by the employer during the year in the following format:

- Amount of tax deposited (positive real numbers with maximum two decimal places)
- Date (dd/mm/yyyy)
- Challan Number (5 to 20 characters)

The above details may be entered multiple times. The function will add all the amounts of tax deposited.

Amount of TDS The function will add all the amounts of tax deposited and amount of TDS in total tax deducted.

It is not clear whether the user can make a wrong entry and move ahead to the next entry or he cannot move ahead until he enters a correct entry.

Generate Tax Payable Details

This function calculates the tax payable by the person in the following format:

Taxable income The function will calculate this by taking difference of gross total income in the function Accept Income Details and aggregate deductible amount in the function Accept Savings & Donation Details.

Tax on taxable income The function will calculate this using the appropriate slab of user as given above.

Surcharge The function will calculate the surcharge as if tax on taxable income exceeds Rs 10 lacs, a 10% surcharge is charged.

Education cess The function will calculate the education cess as a 3% of tax on taxable income.

Tax payable The function will sum up the tax on taxable income, surcharge, and education cess.

Relief under section 89 User will enter the amount, if applicable.

Tax payable after relief (if applicable) The function will deduct the relief amount from the tax payable.

Total tax deducted Displayed from the function Accept Tax Deduction Details.

Tax payable/refundable The function will find difference of tax payable and total tax deducted. If difference is positive, then this amount is the net tax to be paid by the person, otherwise the amount is due on the government to be refunded.

User interface requirements and system requirements are not mentioned.

Step

6

Income Tax Calculator SRS ver 3.0

A system is proposed to calculate the income tax of a person residing in India provided his income, savings, status, and donations are known. The system will accept personal details, income details, savings details and calculate total salary, net tax payable, educational cess, and hence total tax payable. The user gets the information about total tax to be paid.

The system will first accept personal details, income, donations, and savings. For donations, it provides a list of categories in which 100% rebate is provided. The user will look for the option provided and informs the system whether the donation lies in that list. If the donation lies in the list, 100% rebate will be provided, otherwise 50%. The system will check whether the savings are less than Rs 1 lac. If yes, then the whole amount will be deducted from the taxable income. Otherwise, Rs 1 lac will be deducted. Then the system will calculate the total tax and checks if it exceeds Rs 10 lacs, a 10% surcharge on the total income tax (not on the total taxable income) is also charged and a 3% of education cess will be charged on the total income tax paid (not on the total taxable income). Finally, the system will show the net tax as per the following details:

Income Tax slabs 2009/2010 for Men

Income: up to 1.5 lacs	NO TAX
Income : 1.5 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Income Tax slabs 2009/2010 for Women

Income : up to 1.8 lacs	NO TAX
Income : 1.8 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Income Tax slabs 2009/2010 for Senior Citizen

Income : up to 2.25 lacs	NO TAX
Income : 2.25 lacs to 3 lacs	10%
Income : 3 lacs to 5 lacs	20%
Income : above 5 lacs	30%

Donations with 100% rebate

- The Prime Minister's National Relief Fund.
- The Prime Minister's Armenia Earthquake Relief Fund.
- The Africa (Public Contributions-India) Fund.
- The National Foundation for Communal Harmony.
- A University or any educational institution of national eminence as maybe approved by the prescribed authority. In case of any technical institution of national eminence, the prescribed authority is the Director General (Income-Tax Exemption) in concurrence with the Secretary, All India Council of Technical Education.
- The Maharashtra Chief Minister's Earthquake Relief Fund.
- Any Zila Saksharta Samiti constituted in any district under the chairmanship of the Collector of that district for the purpose of improvement of primary education in villages and towns in such a district and for literacy and post literacy activities.
- The National Blood Transfusion Council or any State Blood Transfusion council whose sole objective is the control, supervision, regulation, or encouragement in India of the services related to operation and requirements of blood banks.
- Any fund set up by a State Government to provide medical relief to the poor.
- The Army Central Welfare Fund or the Indian Naval Benevolent Fund or the Air Force Central Welfare Fund established by the armed forces of the

Union for the welfare of the past and present members of such forces or their dependants.

- The Andhra Pradesh Chief Minister's Cyclone Relief Fund, 1996.
- The National Illness Assistance Fund.
- The Chief Minister's Relief Fund or the Lieutenant Governor's Relief Fund in any State or Union Territory.
- The Government, or any local authority, institution or association as maybe approved by the Central Government for the purpose of promoting family planning.

Functional Requirements

Accept Personal Details (APD)

The function will accept the following details to be entered by the user. The user cannot move to the next entry unless he enters the correct entry.

- Name (3 to 15 alphabets with spaces in between)
- Date of Birth (dd/mm/yyyy)
- Permanent address (3 to 30 characters) The allowed characters are alphabets, digits, spaces, and commas only.
- Sex (M/F one alphabet only)
- Status: Salaried or not (Y/N one alphabet only)

If the user enters the answer Y (Yes) to the status entry, then the function will display the following three entries, otherwise it will not.

- Designation (if salaried) (3 to 15 alphabets)
- Name of the employer (if salaried) (3 to 25 alphabets with spaces)
- Address of the employer (if salaried) (3 to 30 characters)
- PAN number (10 characters consisting alphabets and digits 0–9)
- TDS circle where annual return/statement under section 206 is to be filed (3 to 15 alphabets with spaces)
- Period: From (dd/mm/yyyy)
  To (dd/mm/yyyy)
  Assessment year (yyyy-yy)

Accept Income Details (AID)

The function will enquire whether the user is a salaried person or has some other source of income. If the user is not a salaried person, the system will ask for the source of income. The user may enter various types of source of incomes as given below. The user cannot move to the next entry unless he enters the correct entry.

Source of Income: (3 to 20 alphabets with spaces)

Amount: (positive real numbers with maximum two decimal places)

The function will aggregate all the amounts of income as gross total income. If the person is salaried, the function asks for the following details:

1. **Gross Salary**
   (a) Salary as per the provisions contained in the section 17(1)
   (b) Value of the perquisites under section 17(2) (As per form number 12BA, wherever applicable)
   (c) Profits in lieu of salary under section 17(3) (As per form number 12BA, wherever applicable)
   (d) Total (to be calculated by this function)
2. **Less allowance to the extent exempt under section 10**
   This function will add the exempted allowances.
3. **Balance**
   This function will calculate the difference of the gross salary and the exempted allowances.
4. **Deductions**
   - Entertainment allowance (EA)
   - Tax on employment (TE)
5. **Aggregate**
   This function will calculate the aggregate of the deductions entered above.
6. **Income Chargeable Under the Head 'Salaries'**
   The function will calculate the difference of item 3 – item 5.

7. **Add**
   Any other item reported by the employee
   User may enter multiple incomes. The function will add all these incomes.
8. **Gross Total Income**
   The function will add item 6 and 7.
   All the amounts will be positive real numbers with maximum 2 decimal places.

Accept Savings & Donations Details (ASD)

The function will ask the user to enter the total savings and the donations in the following format.

Saving Type: (3 to 20 alphabets with spaces)

Deductible amount: (positive real numbers with maximum two decimal places)

The user may enter multiple savings. The function will add all the deductible amounts in aggregate deductible amount.

The user cannot move to the next entry unless he enters the correct entry.

Accept Tax Deduction Details

If the person is salaried, then this function will accept the details if tax deducted by the employer during the year in the following format:

- Amount of TDS: (positive real numbers with maximum two decimal places)
- Amount of tax deposited (positive real numbers with maximum two decimal places)
- Date (dd/mm/yyyy)
- Challan Number (5 to 20 characters)

The above details may be entered multiple times. The function will add all the amounts of tax deposited and amount of TDS in total tax deducted.

The user cannot move to the next entry unless he enters the correct entry.

Generate Tax Payable Details

This function calculates the tax payable by the person in the following format:

Taxable income Function will calculate this by taking difference of gross total income in the function Accept Income Details and aggregate deductible amount in the function Accept Savings & Donation Details.

Tax on taxable income Function will calculate this using the appropriate slab of user as given above.

Surcharge Function will calculate the surcharge as if tax on taxable income exceeds Rs 10 lacs, a 10% surcharge is charged.

Education cess Function will calculate the education cess as a 3% of tax on taxable income.

Tax payable Function will sum up tax on taxable income, surcharge, and education cess.

Total tax deducted Displayed from the function Accept Tax Deduction Details.

Tax payable/refundable The function will find difference of tax payable and total tax deducted. If difference is positive, then this amount is the net tax to be paid by the person, otherwise the amount is due on the government to be refunded.

User Interface Requirements

Personal Detail Screen

The personal detail screen is displayed wherein the user can enter his various details as given below:

- Name
- Date of Birth
- Permanent address
- Sex (M/F)
- Status (Salaried or not): Y/N

If the user presses Y, then the following screen is displayed:

- Designation
- Name of the employer
- Address of the employer
- PAN number
- TDS circle where annual return/statement under section 206 is to be filed
- Period: From (dd/mm/yyyy)
  To (dd/mm/yyyy)
  Assessment year (yyyy-yy)

If the user presses N, then the following screen is displayed:

- PAN number
- TDS circle where annual return/statement under section 206 is to be filed
- Period: From (dd/mm/yyyy)
  To (dd/mm/yyyy)
  Assessment year (yyyy-yy)

Income Details Screen

The income detail screen is displayed wherein the user can enter his various details as given below.

If the user is not a salaried person, the system will ask for the source of income in the following screen:

- Source of Income:
- Amount:
- Enter more? (Y/N)

If the person is salaried, the function asks for the following details:

1. **Gross Salary**
   (a) Salary as per the provisions contained in the section 17(1)
   (b) Value of the perquisites under section 17(2) (As per form number 12BA, wherever applicable)
   (c) Profits in lieu of salary under section 17(3) (As per form number 12BA, wherever applicable)
   (d) Total (to be calculated by this function)
2. **Allowance to the extent exempt under section 10**
   - Enter more? (Y/N)
3. **Deductions**
   - Entertainment allowance (EA)
   - Tax on employment (TE)
4. **Income Chargeable Under the Head 'Salaries'**
   - Displayed by the system
5. **Any other item reported by the employee**
   - Enter Income:
   - Enter more? (Y/N)
6. **Gross Total Income:** Displayed by the system

Savings & Donations Details Screen

The saving detail screen is displayed wherein the user can enter his various details as given below:

Saving Type: (3 to 20 alphabets with spaces)

Deductible amount: Enter more? (Y/N)

Tax Deduction Details Screen

If the person is salaried, then tax deducted by the employer during the year in the following format is entered:

- Amount of tax deposited
- Date
- Challan Number
- Enter more? (Y/N)
- Amount of TDS:

Tax Payable Detail Screen

This screen is the report screen generated by the system, not modifiable by the user.

Step

7

Black-Box Testing on Units/Modules of Income Tax Calculator SRS ver 3.0

Black-Box Testing on various modules of Income Tax Calculator SRS ver 3.0 is presented here. The details of the various modules have been reproduced to understand the functionality of the module and later, the black-box test cases of modules have been given. The reader is advised to execute these test cases on the running executable application provided in the CD.

Accept Personal Details (APD)

The function will accept the following details to be entered by the user. The user cannot move to the next entry unless he enters the correct entry.

- Name (3 to 15 alphabets with spaces in between)
- Date of Birth (dd/mm/yyyy)
- Permanent address (3 to 30 characters) The allowed characters are alphabets, digits,
- Spaces, and commas only.
- Sex (M/F one alphabet only)
- Status: Salaried or not (Y/N one alphabet only)

If the user enters the answer Y (Yes) to the status entry, then the function will display the following three entries, otherwise it will not.

- Designation (if salaried) (3 to 15 alphabets)
- Name of the employer (if salaried) (3 to 25 alphabets with spaces)
- Address of the employer (if salaried) (3 to 30 characters)
- PAN number (10 characters consisting alphabets and digits 0–9)
- TDS circle where annual return/statement under section 206 is to be filed (3 to 15 alphabets with spaces)
- Period: From (dd/mm/yyyy)
- To (dd/mm/yyyy)
- Assessment year (yyyy-yy)

This module will be tested with equivalence class partitioning methods.

Using the information of the module, the following equivalence classes are generated:

C1 = {3 ≤ Name ≤ 15}

C2 = {Name < 3}

C3 = {Name > 15}

C4 = {Name: Any invalid character other than alphabets and spaces between the alphabets}

C5 = {Name: Blank}

C6 = {Date of Birth: digits only}

C7 = {Date of Birth: Any invalid character other than digit}

C8 = {Date of Birth: Blank}

C9 = {3 ≤ Permanent address ≤ 30}

C10 = {Permanent address < 3}

C11 = {Permanent address > 30}

C12 = {Permanent address: Any invalid character other than alphabets, digits, and spaces between them}

C13 = {Permanent address: Blank}

C14 = {Sex: M/F}

C15 = {Sex: any character other than M/F}

C16 = {Sex: Blank}

C17 = {Status: Y/N}

C18 = {Status: any character other than Y/N}

C19 = {Status: Blank}

C20 = {3 ≤ Designation ≤ 15}

C21 = {Designation < 3}

C22 = {Designation > 15}

C23 = {Designation: Any invalid character other than alphabets and spaces between the alphabets}

C24 = {Designation : Blank}

C25 = {3 ≤ Name of Employer ≤ 25}

C26 = {Name of Employer < 3}

C27 = {Name of Employer > 25}

C28 = {Name of Employer: Any invalid character other than alphabets and spaces between the alphabets}

C29 = {Name of Employer: Blank}

C30 = {Address of the employer < 3}

C31 = {Address of the employer > 30}

C32 = {Address of the employer: Any invalid character other than alphabets, digits and spaces between them}

C33 = {Address of the employer: Blank}

C34 = {PAN number: 10 characters consisting of alphabets and digits only}

C34 = {PAN number < 10}

C34 = {PAN number > 10}

C34 = {PAN number: 10 characters consisting of any invalid character other than alphabets and digits only}

C35 = {PAN number: Blank}

C36 = {3 ≤ TDS Circle ≤ 15}

C37 = {TDS Circle < 3}

C38 = {TDS Circle > 15}

C39 = {TDS Circle: Any invalid character other than alphabets and spaces between the alphabets}

C40 = {TDS Circle: Blank}

C41 = {Period From: digits only}

C42 = {Period From: Any invalid character other than digit}

C43 = {Period From: Blank}

C44 = {Period To: digits only}

C45 = {Period To: Any invalid character other than digit}

C46 = {Period To: Blank}

After preparing the classes for this function, one test case per class should be designed as given below:

Test case ID	Class covered	Name	Date of Birth	Permanent Address	...	Expected Output
APD1	C1	Harish				Normal behaviour
APD2	C2	Na				Name too short
APD3	C3	Abdul Ghaffar Khan				Name too long
APD4	C4	;'.,sfh';.,s				Name should contain only alphabets and spaces.
APD5	C5					Please enter a valid name entry.

APD6	C6	Harish	01/07/1984			Normal behaviour
APD7	C7	Harish	Ab/07/qw12			Date should contain only digits.
APD8	C8	Harish				Please enter a valid Date entry.
....	...	...	...	...	...	...

Accept Income Details (AID)

The function will enquire whether the user is a salaried person or has some other source of income. If the user is not a salaried person, the system will ask for the source of income. The user may enter various types of source of incomes as given below. The user cannot move to the next entry unless he enters the correct entry.

Source of Income: *(3 to 20 alphabets with spaces)*

Amount: *(positive real numbers with maximum two decimal places)*

These two entries can be tested with equivalence class partitioning methods. The classes are:

C1 = {3 ≤ Source of Income ≤ 20}

C2 = {Source of Income < 3}

C3 = {Source of Income > 20}

C4 = {Source of Income: Blank}

C5 = {Amount: positive real numbers with maximum two decimal places}

C6 = {Amount: negative number or characters other than digit}

C7 = {Amount: Blank }

After preparing the classes, one test case per class should be designed as given below:

Test Case ID	Class covered	Source of Income	Amount	Expected Output
AID1	C1	Agriculture		Normal behaviour
AID2	C2	Tc		Entry too short
AID3	C3	Income from consultancy		Entry too long
AID4	C4			Please enter the source of income
AID5	C5	Agriculture	40000.00	Normal behaviour
AID6	C6	Agriculture	Rs 23000.00	Please enter the positive real numbers only
AID7	C7	Agriculture		Please enter amount

The function will aggregate the total amount of income as gross total income.

If the person is salaried, the function asks for the following details:

1. **Gross Salary**
   (a) Salary as per the provisions contained in the section 17(1)
   (b) Value of the perquisites under section 17(2) (As per form number 12BA, wherever applicable)
   (c) Profits in lieu of salary under section 17(3) (As per form number 12BA, wherever applicable)
   (d) Total (to be calculated by this function)

The entries, a, b, c can also be tested with equivalence classes.

C8 = {a: valid entry}

C9 = {a: invalid entry}

C10 = {a:blank}

C11 = {b: valid entry}

C12 = {b: invalid entry}

C13 = {b:blank}

C14 = {c: valid entry}

C15 = {c: invalid entry}

C16 = {c: blank}

Therefore, the test cases will be as follows.

Test Case ID	Class covered	a	b	c	Expected Output
AID8	C8	2000.00			Normal Behaviour
AID9	C9	–1200			Please enter a valid entry.
AID10	C10				Please enter a valid amount.
AID11	C11	2000.00	1000.00		Normal Behaviour
AID12	C12	2000.00	Rs 200		Please enter a valid entry.
AID13	C13	2000.00			Please enter a valid amount.
AID14	C14	2000.00	1000.00	3000.00	Normal Behaviour
AID15	C15	2000.00	1000.00	Rs 200	Please enter a valid entry.
AID16	C16	2000.00	1000.00		Please enter a valid amount.

2. **Less allowance to the extent exempt under section 10**

   This function will add the exempted allowances.

3. **Balance**

   This function will calculate the difference of the gross salary and the exempted allowances.

4. **Deductions**
   - Entertainment allowance (EA)
   - Tax on employment (TE)

These two entries can be tested with equivalence class partitioning methods. The classes are:

C17 = {EA: valid entry}
C18 = {EA: invalid entry}
C19 = {EA: Blank}
C20 = {TE: valid entry}
C21 = {TE: invalid entry}
C22 = {TE: Blank}

The test cases will be as follows:

Test Case ID	Class covered	EA	TE	Expected Output
AID17	C17	2000.00		Normal behaviour
AID18	C18	–200		Please enter a valid amount.
AID19	C19			Please enter a valid amount.
AID20	C20	2000.00	100.00	Normal behaviour
AID21	C21	2000.00	@12.00	Please enter a valid amount.
AID22	C22	2000.00		Please enter a valid amount.

5. **Aggregate**
   This function will calculate the aggregate of the deductions entered above.
6. **Income Chargeable Under the Head 'Salaries'**
   The function will calculate the difference of item 3 – item 5.
7. **Add**
   Any other item reported by the employee
   User may enter multiple incomes. The function will add all these incomes.

This entry can be tested with equivalence class partitioning methods. The classes are:

C23 = {Income: valid entry}
C24 = {Income: invalid entry}
C25 = {Income: Blank}

The test cases will be:

Test Case ID	Class covered	Income	Expected Output
AID23	C23	200.00	Normal behaviour
AID24	C24	–1000	Please enter a valid amount.
AID25	C25		Please enter a valid amount.

8. **Gross Total Income**
   The function will add item 6 and 7.

All the amounts will be positive real numbers with a maximum of two decimal places.

Accept Savings & Donations Details (ASD)

The function will ask the user to enter the total savings and the donations in the following format.

Saving Type: (3 to 20 alphabets with spaces)

Deductible amount: (positive real numbers with maximum two decimal places)

The user may enter multiple savings. The function will add all the deductible amounts in aggregate deductible amount.

The user cannot move to the next entry unless he enters the correct entry.

These two entries can be tested with equivalence class partitioning methods. The classes are:

C1 = {3 ≤ Saving Type ≤ 20}

C2 = {Saving Type < 3}

C3 = {Saving Type > 20}

C4 = {Saving Type: Blank}

C5 = {Deductible amount: positive real numbers with maximum two decimal places}

C6 = {Deductible amount: negative number or characters other than digit}

C7 = {Deductible amount: Blank }

After preparing the classes, one test case per class should be designed as given below:

Test Case ID	Class covered	Saving Type	Deductible Amount	Expected Output
ASD1	C1	LIC		Normal behaviour
ASD2	C2	MF		Entry too short
ASD3	C3	National saving certificate		Entry too long
ASD4	C4			Please enter the saving type
ASD5	C5	LIC	2000.00	Normal behaviour
ASD6	C6	LIC	Rs 23000.00	Please enter the positive real numbers only
ASD7	C7	LIC		Please enter amount

Accept Tax Deduction Details (ATD)

If the person is salaried, then this function will accept the details if tax deducted by the employer during the year is in the following format:

- Amount of tax deposited (positive real numbers with maximum two decimal places)
- Date (dd/mm/yyyy)
- Challan Number (5 to 20 characters)

The above details may be entered multiple times. The function will add the total amounts of tax deposited.
The user cannot move to the next entry unless he enters the correct entry.

Amount of TDS: (positive real numbers with maximum two decimal places)

Testing of these entries can be done with equivalence class partitioning methods. The classes are:

C1 = {Amount of tax deposited: valid entry}

C2 = {Amount of tax deposited: invalid entry}

C3 = {Amount of tax deposited: Blank}

C4 = {Date : digits only}

C5 = {Date : Any invalid character other than digit}

C6 = {Date : Blank}

C7 = {5 ≤ Challan Number ≤ 20}

C8 = {Challan Number < 5}

C9 = {Challan Number > 20}

C10 = {Challan Number: Blank}

Test Case ID	Class covered	Amount of tax deposited	Date	Challan Number	Expected Output
ATD1	C1	2000			Normal behaviour
ATD2	C2	Rs 2000			Please enter the valid amount.
ATD3	C3				Blank Entry. Please enter the valid amount.
ATD4	C4	2000	12/02/2009		Normal behaviour
ATD5	C5	2000	12/feb/2009		Please enter a valid date.
ATD6	C6	2000			Blank entry. Please enter a valid date.
ATD7	C7	2000	12/02/2009	SDE345	Normal behaviour
ATD8	C8	2000	12/02/2009	SDE	Challan number too short
ATD9	C9	2000	12/02/2009	Sdefrtg5667 89asdf5678	Challan number too long
ATD10	C10	2000	12/02/2009		Blank entry. Please enter challan number.

The testing of date field can be done separately with the help of BVA technique. So, now we will test this field with the following specifications:

$1 \leq mm \leq 12$

$1 \leq dd \leq 31$

$2009 \leq yyyy \leq 2099$

Test Cases Using BVC

	Month	Day	Year
Min value	1	1	2009
Min^+ value	2	2	2010
Max value	12	31	2099
Max^- value	11	30	2098
Nominal value	6	15	2060

Using these values, test cases can be designed as shown below:

Test Case ID	Month	Day	Year	Expected Output
1	1	15	2060	Normal behaviour
2	2	15	2060	Normal behaviour
3	11	15	2060	Normal behaviour
4	12	15	2060	Normal behaviour
5	6	1	2060	Normal behaviour
6	6	2	2060	Normal behaviour
7	6	30	2060	Normal behaviour
8	6	31	2060	Invalid input
9	6	15	2009	Normal behaviour
10	6	15	2010	Normal behaviour
11	6	15	2098	Normal behaviour
12	6	15	2099	Normal behaviour
13	6	15	2060	Normal behaviour

Test Cases Using Robust Testing

	Month	Day	Year
Min^- value	0	0	2008
Min value	1	1	2009
Min^+ value	2	2	2010
Max value	12	31	2099
Max^- value	11	30	2098
Max^+ value	13	32	3000
Nominal value	6	15	2060

Using these values, test cases can be designed as shown below:

Test Case ID	Month	Day	Year	Expected Output
1	0	15	2060	Invalid date
2	1	15	2060	Normal behaviour
3	2	15	2060	Normal behaviour
4	11	15	2060	Normal behaviour
5	12	15	2060	Normal behaviour
6	13	15	2060	Invalid date
7	6	0	2060	Invalid date
8	6	1	2060	Normal behaviour

9	6	2	2060	Normal behaviour
10	6	30	2060	Normal behaviour
11	6	31	2060	Invalid input
12	6	32	2060	Invalid date
13	6	15	2008	Invalid date
14	6	15	2009	Normal behaviour
15	6	15	2010	Normal behaviour
16	6	15	2098	Normal behaviour
17	6	15	2099	Normal behaviour
18	6	15	3000	Invalid date
19	6	15	2060	Normal behaviour

Test Cases Using Worst Case Testing

	Month	Day	Year
Min value	1	1	2009
Min^+ value	2	2	2010
Max value	12	31	2099
Max^- value	11	30	2098
Nominal value	6	15	2060

Using these values, test cases can be designed as shown below:

Test Case ID	Month	Day	Year	Expected Output
1	1	1	2009	Normal behaviour
2	1	1	2010	Normal behaviour
3	1	1	2060	Normal behaviour
4	1	1	2098	Normal behaviour
5	1	1	2099	Normal behaviour
6	1	2	2009	Normal behaviour
7	1	2	2010	Normal behaviour
8	1	2	2060	Normal behaviour
9	1	2	2098	Normal behaviour
10	1	2	2099	Normal behaviour
11	1	15	2009	Normal behaviour
12	1	15	2010	Normal behaviour
13	1	15	2060	Normal behaviour
14	1	15	2098	Normal behaviour
15	1	15	2099	Normal behaviour
16	1	30	2009	Normal behaviour
17	1	30	2010	Normal behaviour
18	1	30	2060	Normal behaviour
19	1	30	2098	Normal behaviour
20	1	30	2099	Normal behaviour
21	1	31	2009	Normal behaviour
22	1	31	2010	Normal behaviour

23	1	31	2060	Normal behaviour
24	1	31	2098	Normal behaviour
25	1	31	2099	Normal behaviour
26	2	1	2009	Normal behaviour
27	2	1	2010	Normal behaviour
28	2	1	2060	Normal behaviour
29	2	1	2098	Normal behaviour
30	2	1	2099	Normal behaviour
31	2	2	2009	Normal behaviour
32	2	2	2010	Normal behaviour
33	2	2	2060	Normal behaviour
34	2	2	2098	Normal behaviour
35	2	2	2099	Normal behaviour
36	2	15	2009	Normal behaviour
37	2	15	2010	Normal behaviour
38	2	15	2060	Normal behaviour
39	2	15	2098	Normal behaviour
40	2	15	2099	Normal behaviour
41	2	30	2009	Invalid date
42	2	30	2010	Invalid date
43	2	30	2060	Invalid date
44	2	30	2098	Invalid date
45	2	30	2099	Invalid date
46	2	31	2009	Invalid date
47	2	31	2010	Invalid date
48	2	31	2060	Invalid date
49	2	31	2098	Invalid date
50	2	31	2099	Invalid date
51	6	1	2009	Normal behaviour
52	6	1	2010	Normal behaviour
53	6	1	2060	Normal behaviour
54	6	1	2098	Normal behaviour
55	6	1	2099	Normal behaviour
56	6	2	2009	Normal behaviour
57	6	2	2010	Normal behaviour
58	6	2	2060	Normal behaviour
59	6	2	2098	Normal behaviour
60	6	2	2099	Normal behaviour
61	6	15	2009	Normal behaviour
62	6	15	2010	Normal behaviour
63	6	15	2060	Normal behaviour
64	6	15	2098	Normal behaviour
65	6	15	2099	Normal behaviour

66	6	30	2009	Normal behaviour
67	6	30	2010	Normal behaviour
68	6	30	2060	Normal behaviour
69	6	30	2098	Normal behaviour
70	6	30	2099	Normal behaviour
71	6	31	2009	Invalid date
72	6	31	2010	Invalid date
73	6	31	2060	Invalid date
74	6	31	2098	Invalid date
75	6	31	2099	Invalid date
76	11	1	2009	Normal behaviour
77	11	1	2010	Normal behaviour
78	11	1	2060	Normal behaviour
79	11	1	2098	Normal behaviour
80	11	1	2099	Normal behaviour
81	11	2	2009	Normal behaviour
82	11	2	2010	Normal behaviour
83	11	2	2060	Normal behaviour
84	11	2	2098	Normal behaviour
85	11	2	2099	Normal behaviour
86	11	15	2009	Normal behaviour
87	11	15	2010	Normal behaviour
88	11	15	2060	Normal behaviour
89	11	15	2098	Normal behaviour
90	11	15	2099	Normal behaviour
91	11	30	2009	Normal behaviour
92	11	30	2010	Normal behaviour
93	11	30	2060	Normal behaviour
94	11	30	2098	Normal behaviour
95	11	30	2099	Normal behaviour
96	11	31	2009	Invalid date
97	11	31	2010	Invalid date
98	11	31	2060	Invalid date
99	11	31	2098	Invalid date
100	11	31	2099	Invalid date
101	12	1	2009	Normal behaviour
102	12	1	2010	Normal behaviour
103	12	1	2060	Normal behaviour
104	12	1	2098	Normal behaviour
105	12	1	2099	Normal behaviour
106	12	2	2009	Normal behaviour
107	12	2	2010	Normal behaviour

108	12	2	2060	Normal behaviour
109	12	2	2098	Normal behaviour
110	12	2	2099	Normal behaviour
111	12	15	2009	Normal behaviour
112	12	15	2010	Normal behaviour
113	12	15	2060	Normal behaviour
114	12	15	2098	Normal behaviour
115	12	15	2099	Normal behaviour
116	12	30	2009	Normal behaviour
117	12	30	2010	Normal behaviour
118	12	30	2060	Normal behaviour
119	12	30	2098	Normal behaviour
120	12	30	2099	Normal behaviour
121	12	31	2009	Normal behaviour
122	12	31	2010	Normal behaviour
123	12	31	2060	Normal behaviour
124	12	31	2098	Normal behaviour
125	12	31	2099	Normal behaviour

Generate Tax Payable Details

This function calculates the tax payable by the person in the following format:

Taxable income The function will calculate this by taking the difference of gross total income in the function Accept Income Details and aggregate deductible amount in the function Accept Savings & Donation Details.

Tax on taxable income The function will calculate this using the appropriate slab of user as given above.

The following conditions are there for calculating the appropriate tax slab of a person:

- Is Sex Male?
- Age > 65?
- Income: up to 1.5 lacs
- Income: up to 1.8 lacs
- Income : 1.5 lacs to 3 lacs
- Income : 1.8 lacs to 3 lacs
- Income : 3 lacs to 5 lacs
- Income : above 5 lacs
- Income : up to 2.25 lacs
- Income : 2.25 lacs to 3 lacs

The following outputs will be there:

- No tax
- 10%
- 20%
- 30%

The test cases for these conditions and outputs can be designed using a decision table, as given below:

		R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12
Condition Stub	C1: Is Sex Male?	T	T	T	T	F	F	F	F	I	I	I	I
	C2: Age > 60?	F	F	T	F	F	F	F	F	T	T	T	T
	C3: Income: up to 1.5 lacs	T	F	F	F	I	F	F	F	I	F	F	F
	C4: Income: up to 1.8 lacs	F	I	F	F	T	F	F	F	I	F	F	F
	C5: Income : 1.5 lacs to 3 lacs	F	T	F	F	F	I	F	F	I	I	F	F
	C6: Income : 1.8 lacs to 3 lacs	F	F	F	F	F	T	F	F	I	I	F	F
	C7: Income : 3 lacs to 5 lacs	F	F	T	F	F	F	T	F	F	F	T	F
	C8: Income : above 5 lacs	F	F	F	T	F	F	F	T	F	F	F	T
	C9: Income : up to 2.25 lacs	F	I	F	F	F	I	F	I	T	F	F	F
	C10: Income : 2.25 lacs to 3 lacs	F	I	F	F	F	I	F	F	F	T	F	F
Action Stub	A1: No Tax	X				X				X			
	A2: 10%		X				X				X		
	A3: 20%			X				X				X	
	A4: 30%				X				X				X

Surcharge The function will calculate the surcharge as: If tax on taxable income exceeds Rs 10 lacs, a 10% surcharge is charged.

Education cess The function will calculate the education cess as a 3% of tax on taxable income.

Tax payable The function will sum up the tax on taxable income, surcharge, and education cess.

Relief under section 89 User will enter the amount, if applicable.

Tax payable after relief (if applicable) The function will deduct relief amount from tax payable.

Total Tax deducted Displayed from the function Accept Tax Deduction Details.

Tax payable/refundable The function will find the difference of tax payable and the total tax deducted. If difference is positive, then this amount is the net tax to be paid by the person, otherwise the amount is due on the government to be refunded.

Step

8

White-Box Testing on Units/Modules of Income Tax Calculator

Here, we will discuss how the modules of Income Tax Calculator is tested using white-box testing techniques. The modules of TaxCalculator.h (refer CD) have been reproduced here for understanding their functionality and coding details. The reader is advised to refer to the full code for implementation details of the application and execute these test cases on the running executable application provided in the CD.

Basis Path Testing on income_details_non_sal()

Source Code of income_details_non_sal()

```
float income_details_non_sal()
     {
1'   char source[20]="abc";
2'   float amount,total =0;
3'   int flag1=1,flag2=1,i;
4'   char income_ch='y';

1    while((income_ch=='y')||(income_ch=='Y'))
2        {
3        while(flag1==1)
4            {
5            printf("\nEnter SOURCE\t:");
6            gets(source);
7            for(i=0;i<strlen(source);i++)
8                {
9                 if(((toascii(source[i]) >= 65) && (toascii(source[i])
                  <= 122)) || (toascii(source[i]) == 32))
10                       {
11                       flag1=0;
12                       }
```

```
                    else
                    {
                    printf("\nSource can contain only charcter Error at
                          position number %d",i);
                    flag1=1;
                    break;
                    }
                }//end for
                if((strlen(source)<3)||(strlen(source)>20))
                {
                printf("\nSource can contain a max of 20 characters");
                flag1=1;
                }
}
while(flag2==1)
{
printf("\nEnter Amount\t:");
scanf("%f",&amount);
if(amount>0)
          {
          flag2=0;
          }
     else
          {
          printf("\nAmount cannot be less than or equal to 0");
          flag2=1;
          }
}
printf("\n\nPress any key to proceed");
getch();
clrscr();
patt("INCOME Details");
printf("\nSOURCE\t:%s",source);
printf("\nAMOUNT\t:%f",amount);
total=total+amount;
printf("\nDo you want to enter more(y/n)\t:");
income_ch=getche();
flag1=1;
flag2=1;
}
printf("\nTotal\t\t:%f",total);
return(total);
}
```

DD Graph for Module income_details_non_sal()

Cyclomatic Complexity of income_details_non_sal()

1. $V(G) = e - n + 2P$
   $= 29 - 23 + 2$
   $= 8$
2. $V(G) =$ Number of predicate nodes + 1
   $= 7 + 1$
   $= 8$
3. $V(G) =$ No. of regions
   $= 8$

Independent Paths of income_details_non_sal()

Since the cyclomatic complexity of the graph is 8, there will be 8 independent paths in the graph, as shown below:

1. $N_1 N_2 N_{23}$
2. $N_1 N_2\ N_3\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
3. $N_1 N_2\ N_3\ N_4\ N_5\ N_6\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
4. $N_1 N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_{10}\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
5. $N_1 N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_9\ N_{11}\ N_6\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
6. $N_1 N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_9\ N_{11}\ N_6\ N_{12}\ N_{13}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
7. $N_1 N_2\ N_3\ N_4\ N_{15}\ N_{16}\ N_{17}\ N_{19}\ N_{20}\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
8. $N_1 N_2\ N_3\ N_4\ N_{15}\ N_{16}\ N_{17}\ N_{18}\ N_{20}\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$

Test Case Design on income_details_non_sal() from the list of Independent Paths

Test Case ID	Inputs		Expected Output	Independent paths covered by Test Case
	Source	Amount		
1	Agriculture	400000	Source Agriculture Amount 400000 Do you want to enter more(y/n) Y	1, 2, 3, 5, 8
	Others	100000	Source Others Amount 100000 Do you want to enter more(y/n) N	
			Total 500000	
2	1234		Source can contain only character.	1, 2, 3, 4
3	Agriculture and others		Source can contain a max of 20 characters.	1, 2, 3, 6
4	Agriculture	0	Amount cannot be less than or equal to 0	1, 2, 3, 5, 7

Data Flow Testing on income_details_non_sal()

Definition Nodes and Usage Nodes

Variable	Defined At	Used At
source	1', 6	7, 9, 20, 44
amount	29	30, 45, 46
total	2', 46	46, 52
flag1	3', 11, 16, 23, 49	3
flag2	3', 32, 37, 50	26
i	7	7, 9, 15
income_ch	4', 48	1

du and dc paths

Variable	du Path(beg-end)	dc?
source	1'–7	No
	1'–9	No
	1'–20	No
	1'–44	No
	6–7	Yes
	6–9	Yes
	6–20	Yes
	6–44	Yes
amount	29–30	Yes
	29–45	Yes
	29–46	Yes
total	2'–46	Yes
	2'–52	No
	46–46	Yes
	46–52	Yes
flag1	3'–3	Yes
	11–3	No
	16-3	No
	23–3	Yes
	49–3	Yes

flag2	3'–26	Yes
	32–26	Yes
	37–26	Yes
	50–26	Yes
i	7–7	Yes
	7–9	Yes
	7–15	Yes
income_ch	4'–1	Yes
	48–1	Yes

Basis Path Testing on income_details_sal()

Source code of income_details_sal()

```
double income_details_sal()
{
1'   float t_d, d1, d2, sal1, sal2, sal3, t_sal, sal_all, sal_all_tot=0, ei,
     t_ei=0, net_t_sal=0, bal;
2'   char sal_ch='y',ei_ch='y';
3'   int f1=1,f2=1,f3=1,f4=1;
4'   double gross;

1    while(f2==1)
2         {
3         printf("\n1.\tGROSS SALARY\t:");
4         printf("\n\ta) Salary as per the provisions contained in the sec-
               tion 17(1)\t:");
5         scanf("%f",&sal1);
6         printf("\n\tb) Value of the perquisites under section 17(2)\n(As
               per form number 12BA , wherever applicable )\t:");
7         scanf("%f",&sal2);
8         printf("\n\tc) Profits in lieu of salary under section 17(3)\n(As
               per form number 12BA , wherever applicable )\t:");
9         scanf("%f",&sal3);
10        if((sal1<0)||(sal2<0)||(sal3<0))
11             {
12             f2=1;
13             }
14        else
15             {
16             f2=0;
```

```
17          }
18      }
19      t_sal=sal1+sal2+sal3;
20      printf("\n\td)\tTotal\n\t\t\t\t:%f",t_sal);
21      sal_all_tot=0;
22      while((sal_ch=='y')||(sal_ch=='Y'))
23      {
24      while(f1==1)
25          {
26          printf("\n2.\tAllowance to the extent exempt under section
            10\t:");
27          scanf("%f",&sal_all);
28          if(sal_all<0)
29           {
30           printf("\nEnter correct value");
31           }
32          else
33           {
34           f1=0;
35           }
36          }
37      sal_all_tot=sal_all_tot+sal_all;
38      printf("\nEnter more?(Y/N)\t:");
39      sal_ch=getche();
40      if((sal_ch=='y')||(sal_ch=='Y')||(sal_ch=='n')||(sal_ch=='N'))
41           {
42           f1=0;
43           }
44      else
45           {
46           printf("\nPlease enter y or n");
47           }
48      }
49      printf("\nTotal allowance\t\t:%f",sal_all_tot);
50      bal=t_sal-sal_all_tot;
51      printf("\nBalance\t:%f",bal);
52      while(f3==1)
53      {
54      printf("\n3.\tDeductions\t:");
55      printf("\n\tEntertainment allowance(EA)\t:");
56      scanf("%f",&d1);
```

```
57        printf("\tTax on employment (TE)\t:");
58        scanf("%f",&d2);
59        if((d1<0)||(d2<0))
60             {
61             f3=1;
62             }
63        else
64             {
65             f3=0;
66             }
67        }
68        t_d=d1+d2;
69        printf("\nTotal deductions\t:%f",t_d);
70        net_t_sal=bal-t_d;
71        printf("\n4.\tINCOME CHARGABLE UNDER THE HEAD SALARIES'\t:%f",net_t_sal);
72        while((ei_ch=='y')||(ei_ch=='Y'))
73        {
74        while(f4==1)
75        {
76        printf("\n5.\tAny other income reported by the Employee\t:");
77        printf("\n\t\tEnter Income\t:");
78        scanf("%f",&ei);
79        if(ei<0)
80             {
81             f4=1;
82             }
83        else
84             {
85             f4=0;
86             }
87        }
88        t_ei=t_ei+ei;
89        printf("\n\t\tEnter more?(Y/N)\t:");
90        ei_ch=getche();
91        }
92        gross=net_t_sal+t_ei;
93        printf("\n6.Gross Total Income:\t%f",gross);
94        return(gross);
95   }
```

DD Graph for Module income_details_sal()

Cyclomatic Complexity of income_details_sal()

1. $V(G) = e - n + 2P$
   $= 52 - 42 + 2$
   $= 12$
2. $V(G) =$ Number of predicate nodes + 1
   $= 11 + 1$
   $= 12$
3. $V(G) =$ No. of Regions
   $= 12$

Independent Paths of income_details_sal()

Since the cyclomatic complexity of the graph is 12, there will be 12 independent paths in the graph as shown below:

1. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
2. $N_1\ N_2\ N_3\ N_4\ N_6\ N_7\ N_2\ N_8\ N_9\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
3. $N_1\ N_2\ N_3\ N_4\ N_5\ N_7\ N_2\ N_8\ N_9\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
4. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{11}\ N_{12}\ N_{18}\ N_{19}\ N_{22}\ N_{21}\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
5. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{11}\ N_{12}\ N_{13}\ N_{14}\ N_{16}\ N_{17}\ N_{12}\ N_{18}\ N_{19}\ N_{22}\ N_{21}\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
6. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{11}\ N_{12}\ N_{13}\ N_{14}\ N_{15}\ N_{17}\ N_{12}\ N_{18}\ N_{19}\ N_{22}\ N_{21}\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
7. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{11}\ N_{12}\ N_{18}\ N_{19}\ N_{20}\ N_{21}\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
8. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{27}\ N_{28}\ N_{30}\ N_{31}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
9. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{27}\ N_{28}\ N_{29}\ N_{31}\ N_{25}\ N_{26}\ N_{32}\ N_{41}\ N_{42}$
10. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{27}\ N_{28}\ N_{30}\ N_{31}\ N_{25}\ N_{26}\ N_{32}\ N_{33}\ N_{34}\ N_{40}\ N_{32}\ N_{41}\ N_{42}$
11. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{27}\ N_{28}\ N_{30}\ N_{31}\ N_{25}\ N_{26}\ N_{32}\ N_{33}\ N_{34}\ N_{35}\ N_{36}\ N_{38}\ N_{39}\ N_{34}\ N_{40}\ N_{32}\ N_{41}\ N_{42}$
12. $N_1\ N_2\ N_8\ N_9\ N_{10}\ N_{23}\ N_{24}\ N_{25}\ N_{27}\ N_{28}\ N_{30}\ N_{31}\ N_{25}\ N_{26}\ N_{32}\ N_{33}\ N_{34}\ N_{35}\ N_{36}\ N_{37}\ N_{39}\ N_{34}\ N_{40}\ N_{32}\ N_{41}\ N_{42}$

Test Case Design on income_details_sal() from the list of Independent Paths

Test Case ID	Inputs								Expected Output	Independent path covered by Test Case
	Sal1	Sal2	Sal3	Sal_all	Enter more?	EA	TE	Other Income		
1	2000	4000	9000						Total 15000	1, 2, 5, 7, 8, 10, 11
				2000	Y					
				1000	n				Total Allowance 3000 Balance 12000	
						200	300		Total deductions 500 Income under Head Salaries 11500	
								12000	Enter more? Y	
								12000	Enter more? N Gross Total Income: 35500	
2	2000	–300	500							2, 3
3	2000	4000	9000						Total 15000	2, 6
				–300					Enter correct value	
4	2000	4000	9000						Total 15000	2, 4, 5, 7
				1000	t				Please enter y or n	
5	2000	4000	9000						Total 15000	2, 5, 7, 9
				1000	n				Total Allowance 1000 Balance 14000	
						–100	200			
6	2000	4000	9000						Total 15000	2, 5, 7, 8, 10, 12
				2000	Y					
				1000	n				Total Allowance 3000 Balance 12000	
						200	300		Total deductions 500 Income under Head Salaries 11500	
								–1000		

Data Flow Testing on income_details_sal()

Definition nodes and Usage nodes

Variable	Defined At	Used At
t_d	68	69, 70
d1	56	59, 68
d2	58	59, 68
sal1	5	10, 19
sal2	7	10, 19
sal3	9	10, 19
t_sal	19	20
sal_all	27	28, 37
sal_all_tot	21, 37	37, 49, 50
ei	78	79, 88
t_ei	1', 88	88, 92
net_t_sal	1', 70	71, 92
bal	50	70
sal_ch	2', 39	40
ei_ch	2', 90	72
f1	3', 34, 42	24
f2	3', 12, 16	1
f3	3', 61, 65	52
f4	3', 81, 85	74
gross	92	93

du and dc paths

Variable	du Path(beg-end)	dc?
t_d	68–69	Yes
	68–70	Yes
d1	56–59	Yes
	56–68	Yes
d2	58–59	Yes
	58–68	Yes
sal1	5–10	Yes
	5–19	Yes
sal2	7–10	Yes
	7–19	Yes
sal3	9–10	Yes
	9–19	Yes
t_sal	19–20	Yes
sal_all	27–28	Yes
	27–37	Yes
sal_all_tot	21–37	No

	21–49	No
	21–50	No
	37–37	No
	37–49	Yes
	37–50	Yes
ei	78–79	Yes
	78–88	Yes
t_ei	1'–88	No
	1'–92	No
	88–88	No
	88–92	Yes
net_t_sal	1'–71	No
	1'–92	No
	70–71	Yes
	70–92	Yes
bal	50–70	Yes
sal_ch	2'–40	No
	39–40	Yes
ei_ch	2'–72	Yes
	90–72	Yes
f1	3'–24	Yes
	34–24	Yes
	42–24	Yes
f2	3'–1	Yes
	12–1	Yes
	16–1	Yes
f3	3'–52	Yes
	61–52	Yes
	65–52	Yes
f4	3'–74	Yes
	81–74	Yes
gross	85–74	Yes
	92–93	Yes

Basis Path testing on module savings()

Source Code of module savings()

```
float savings()
     {
1    char saving_type[20];
2    float amount,total=0;
3    int flag1=1,flag2=1,i;
4    char sav_ch='y';
5    while((sav_ch=='y')||(sav_ch=='Y'))
6    {
```

```
7    while(flag1==1)
8        {
9        printf("\nEnter Saving type\t:");
10       gets(saving_type);
11       for(i=0;i<strlen(saving_type);i++)
12           {
13            if(((toascii(saving_type[i])>= 65) && (toascii(saving_type[i])
              <= 122)) || (toascii(saving_type[i]) == 32))
14            {
15            flag1=0;
16            }
17           else
18            {
19           printf("\nSaving type can contain only charcter Error at posi-
              tion number %d",i);
20            flag1=1;
21            break;
22            }
23           }
24           if((strlen(saving_type)<3)||(strlen(saving_type)>20))
25           {
26           printf("\nPlease enter between 3 to 20 characters ");
27           flag1=1;
28           }
29       }
30       while(flag2==1)
31       {
32       printf("\nEnter Amount\t:");
33       scanf("%f",&amount);
34       if(amount>0)
35            {
36            flag2=0;
37            }
38           else
39            {
40            printf("\nAmount cannot be less than or equal to 0");
41            flag2=1;
42            }
43       }
44       printf("\n\nPress any key to proceed");
45       getch();
46       clrscr();
47       patt("SAVING Details");
48       printf("\nSAVING TYPE\t:%s",saving_type);
49       printf("\nAMOUNT\t\t\t:%f",amount);
50       total=total+amount;
51       printf("\nDo you want to enter more(y for yes)\t:");
52       sav_ch=getche();
53       flag1=1;
```

```
54        flag2=1;
55        }
56        printf("\nTotal\t\t:%f",total);
57        return(total);
58    }
```

DD Graph for Module savings()

Cyclomatic Complexity of savings()

1. $V(G) = e - n + 2P$
   $= 29 - 23 + 2$
   $= 8$
2. $V(G) =$ Number of predicate nodes + 1
   $= 7 + 1$
   $= 8$
3. $V(G) =$ No. of regions
   $= 8$

Independent Paths of savings()

Since the cyclomatic complexity of the graph is 8, there will be 8 independent paths in the graph, as shown below:

1. $N_1\ N_2\ N_{23}$
2. $N_1\ N_2\ N_3\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
3. $N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
4. $N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_{10}\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
5. $N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8\ N_9\ N_{11}\ N_6\ N_{12}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
6. $N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_{12}\ N_{13}\ N_{14}\ N_4\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
7. $N_1\ N_2\ N_3\ N_4\ N_{15}\ N_{16}\ N_{17}\ N_{19}\ N_{20}\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$
8. $N_1\ N_2\ N_3\ N_4\ N_{15}\ N_{16}\ N_{17}\ N_{18}\ N_{20}\ N_{15}\ N_{21}\ N_{22}\ N_2\ N_{23}$

Test Case Design on savings() from the list of Independent Paths

Test Case ID	Inputs			Expected Output	Independent path covered by Test Case
	Saving Type	Amount	Enter more?		
1	NSC	5000		Saving type NSC Amount 5000	1, 2, 3, 5, 8
			y		
	PPF	1200		Saving type PPF Amount 1200	
			n	Total 6200	
2	123			Saving type can contain only character	2, 3, 4

3	PF			Please enter between 3 to 20 characters	2, 3, 6
4	PPF	0		Amount cannot be less than or equal to 0.	2, 3, 7

Data Flow Testing on module savings()

Definition and Usage nodes

Variable	Defined At	Used At
saving_type	10	11, 13, 24, 48
amount	33	34, 49, 50
total	2, 50	50, 56, 57
flag1	3, 15, 20, 27, 53	7
flag2	3, 36, 41, 54	30
i	11	11, 13, 19
sav_ch	4, 52	5

du and dc paths

Variable	du Path(beg-end)	dc?
saving_type	10–11	Yes
	10–13	Yes
	10–24	Yes
	10–48	Yes
amount	33–34	Yes
	33–49	Yes
	33–50	Yes
total	2–50	No
	50–50	No
	50–56	Yes
	50–57	Yes
flag1	3–7	Yes
	15–7	No
	20–7	No
	27–7	Yes
	53–7	Yes
flag2	3–30	Yes
	36–30	Yes
	41–30	Yes
	54–30	Yes

i	11–11	No
	11–13	Yes
	11–19	Yes
sav_ch	4–5	Yes
	52–5	Yes

Basis Path Testing on module tax_ded()

Source code of tax_ded()

```
float  tax_ded()
     {
1    char challan[20];
2    int flag=1,flag1=1,flag2=1,flag3=1,flag4=1,i,dd,mm,yyyy,choice='y';
3    float amount_tax,tot_amount_tax=0,amount_tds,tot;

4    while((choice=='y')||(choice=='Y'))
5    {   flag=1,flag1=1,flag2=1,flag3=1,flag4=1;
6        while(flag4==1)
7        {
8        printf("\nEnter amount of Tax deposited\t:");
9        scanf("%f",&amount_tax);
10           if(amount_tax<0)
11            {
12            printf("\nAMOUNT CANNOT BE NEGATIVE");
13            flag4=1;
14            }
15           else
16            {
17            flag4=0;
18            }
19       }
20       while(flag3==1)
21       {
22           flag3=0;
23           printf("\nEnter DATE:\t:");
24           printf("\n\tEnter day(dd)\t:");
25           scanf("%d",&dd);
26            if((dd>31)||(dd<=0))
27            {
28            flag3=1;
29            printf("\nWrong date");
30            }
```

```
31          printf("\n\tEnter month\t:");
32          scanf("%d",&mm);
33          if((mm<=0)||(mm>12)||((mm==2)&&(dd>29)))
34           {
35           printf("\nWrong date/month");
36           flag3=1;
37           }
38          printf("\n\tEnter year\t:");
39          scanf("%d",&yyyy);
40          if((yyyy<2009)||(yyyy>2099))
41           {
42           printf("\nPlease enter a year between 2009 and 2099");
43           flag3=1;
44           }
45      }
46      while(flag1==1)
47      {
48          printf("\nEnter Challan Number\t:");
49          gets(challan);
50          for(i=0;i<strlen(challan);i++)
51          {
52          if(((toascii(challan[i]) >= 65) && (toascii(challan[i]) <=
             122)) || (toascii(challan[i]) == 32) || ((toascii(challan[i])
             >= 48) && (toascii(challan[i]) <= 57)))
53           {
54           flag1=0;
55           }
56          else
57           {
58           printf("\nChallan Number can contain only charcter Error at
                position number %d",i);
59           flag1=1;
60           break;
61           }
62          }//end of for
63          if((strlen(challan)<5)||(strlen(challan)>20))
64          {
65          printf("\nPlease enter correct Challan Number having 5 to 20
                characters");
66          flag1=1;
67          }
68      }
69      printf("\nAMOUNT OF TAX DEPOSITED\t:%f",amount_tax);
```

```
70      printf("\nCHALLAN NUMBER\t\t:%s",challan);
71      printf("\nDATE\t\t\t:%d/%d/%d",dd,mm,yyyy);
72      flag=1;
73      while(flag)
74      {
75      printf("\nDo you want to enter more ?(y/n)");
76      choice=getche();
77      if((choice=='y')||(choice=='Y')||(choice=='N')||(choice=='n'))
78          {
79          flag=0;
80          }
81          else
82          {
83          printf("\nPlease Enter y or n");
84          flag=1;
85          }
86      }//end of inner while
87  tot_amount_tax=tot_amount_tax+amount_tax;
88  }
89      while(flag2==1)
90      {
91      printf("\nEnter amount of TDS \t:");
92      scanf("%f",&amount_tds);
93      if(amount_tds<0)
94          {
95          printf("\nTDS AMOUNT CANNOT BE NEGATIVE");
96          flag2=1;
97          }
98      else
99          {
100         flag2=0;
101         }
102     }
103     printf("\n\nPress any key to proceed");
104     getch();
105     clrscr();
106     patt("TAX DEDUCTION");
107     printf("\nAMOUNT OF TAX DEPOSITED\t:%f",tot_amount_tax);
108     printf("\nAMOUNT OF TDS\t:%f",amount_tds);
109     tot=tot_amount_tax+amount_tds;
110     printf("\nTotal\t\t:%f",tot);
111     return(tot);
112     }
```

DD Graph for Module tax_ded()

Cyclomatic Complexity of tax_ded()

1. $V(G) = e - n + 2P$
   $= 61 - 47 + 2$
   $= 16$
2. $V(G) =$ Number of predicate nodes + 1
   $= 15 + 1$
   $= 16$
3. $V(G) =$ No. of regions
   $= 16$

Independent Paths of tax_ded()

Since the cyclomatic complexity of the graph is 16, there will be 16 independent paths in the graph as shown below:

1. $N_1\ N_2\ N_{41}\ N_{47}$
2. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
3. $N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_8\ N_9\ N_4\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
4. $N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_9\ N_4\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
5. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{11}\ N_{12}\ N_{14}\ N_{15}\ N_{17}\ N_{18}\ N_{20}\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
6. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{11}\ N_{12}\ N_{13}\ N_{14}\ N_{15}\ N_{17}\ N_{18}\ N_{20}\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
7. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{11}\ N_{12}\ N_{13}\ N_{14}\ N_{15}\ N_{16}\ N_{17}\ N_{18}\ N_{20}\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
8. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{11}\ N_{12}\ N_{13}\ N_{14}\ N_{15}\ N_{16}\ N_{17}\ N_{18}\ N_{19}\ N_{20}\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
9. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{22}\ N_{23}\ N_{29}\ N_{31}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
10. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{22}\ N_{23}\ N_{29}\ N_{30}\ N_{31}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
11. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{22}\ N_{23}\ N_{24}\ N_{25}\ N_{27}\ N_{28}\ N_{23}\ N_{29}\ N_{31}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
12. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{22}\ N_{23}\ N_{24}\ N_{25}\ N_{26}\ N_{28}\ N_{23}\ N_{29}\ N_{31}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
13. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{34}\ N_{35}\ N_{37}\ N_{38}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
14. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{34}\ N_{35}\ N_{36}\ N_{38}\ N_{33}\ N_{34}\ N_{35}\ N_{37}\ N_{38}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{47}$
15. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{42}\ N_{43}\ N_{45}\ N_{46}\ N_{41}\ N_{47}$
16. $N_1\ N_2\ N_3\ N_4\ N_{10}\ N_{21}\ N_{32}\ N_{33}\ N_{39}\ N_{40}\ N_2\ N_{41}\ N_{42}\ N_{43}\ N_{44}\ N_{46}\ N_{41}\ N_{47}$

Test Case Design on tax_ded() from the list of Independent Paths

Test Case ID	Inputs							Expected Output	Independent path covered by Test Case
	Tax amount	day	month	year	Challan number	Enter more?	TDS		
1	5000	12	12	2009	Dfert56			AMOUNT OF TAX DEPOSITED 5000 CHALLAN NUMBER Dfert56 DATE 12 12 2009	1, 2, 3, 5, 9, 12, 14, 15
						Y			
	2000	13	12	2009	Hgyt65			AMOUNT OF TAX DEPOSITED 2000 CHALLAN NUMBER Hgyt65 DATE 13 12 2009	
						n			
							1200	AMOUNT OF TAX DEPOSITED 7000 AMOUNT OF TDS 1200 Total 8200	
2	–200							AMOUNT CANNOT BE NEGATIVE	2, 3, 4
3	3000	32/0/-1						Wrong date	2, 3, 6
4	3000	31	-1/0/13					Wrong date/ month	2, 3, 7
5	3000	29	2					Wrong date/ month	2, 3, 7
6	3000	31	5	2008				Please enter a year between 2009 and 2099	2, 3, 8

7	4000	31	5	2009	!@#$			Challan number can contain only characters	2, 3, 5, 11
8	4000	31	5	2009	rty			Please enter correct challan number having 5 to 20 characters	2, 3, 5, 10
9	4000	31	5	2009	Rty456	n	–234	TDS AMOUNT CANNOT BE NEGATIVE	2, 3, 5, 12, 14, 16
10	4000	31	5	2009	Rty456	t		Please enter y or n	2, 3, 5, 13

Data Flow Testing on module tax_ded()

Definition and Usage nodes

Variable	Defined At	Used At
challan	49	50, 52, 63, 70
flag	2, 5, 72, 79, 84	73
flag1	2, 5, 54, 59, 66,	46
flag2	2, 5, 96, 100	89
flag3	2, 5, 22, 28, 36, 43	20
flag4	2, 5, 13, 17	6
i	50	50, 52, 58
dd	25	26, 33
mm	32	33
yyyy	39	40
choice	2, 76	77
amount_tax	9	10, 69, 87
tot_amount_tax	3, 87	87, 107, 109
amount_tds	92	93, 108, 109
tot	109	110, 111

du and dc paths

Variable	du Path(beg-end)	dc?
challan	49–50	Yes
	49–52	Yes
	49–63	Yes
	49–70	Yes
flag	2–73	No
	5–73	No
	72–73	Yes
	79–73	Yes
	84–73	Yes
flag1	2–46	No
	5–46	Yes
	54–46	No
	59–46	No
	66–46	Yes
flag2	2–89	No
	5–89	Yes
	96–89	Yes
	100–89	Yes
flag3	2–20	No
	5–20	Yes
	22–20	No
	28–20	No
	36–20	No
	43–20	Yes
flag4	2–6	No
	5–6	Yes
	13–6	Yes
	17–6	Yes
i	50–50	No
	50–52	Yes
	50–58	Yes
dd	25–26	Yes
	25–33	Yes
mm	32–33	Yes
yyyy	39–40	Yes
choice	2–77	No

	76–77	Yes
amount_tax	9–10	Yes
	9–69	Yes
	9–87	Yes
tot_amount_tax	3–87	No
	3–107	No
	3–109	No
	87–87	No
	87–107	Yes
	87–109	Yes
amount_tds	92–93	Yes
	92–108	Yes
	92–109	Yes
tot	109–110	Yes
	109–111	Yes

Basis Path Testing on module male_tax()

Source code of male_tax()

```
float male_tax(float taxable)
        {
        float t;

1       if(taxable<=150000)
2               {
3               return 0;
4               }
5       else if((taxable>150000)&&(taxable<=300000))
6               {
7               t=10*taxable/100;
8               }
9       else if((taxable>300000)&&(taxable<=500000))
10              {
11              t=20*taxable/100;
12              }
13      else
14              {
15              t=30*taxable/100;
16              }
17      return t;
18    }
```

DD Graph for Module male_tax()

Cyclomatic Complexity of male_tax()

1. $V(G) = e - n + 2P$
   $= 8 - 6 + 2$
   $= 4$
2. $V(G) =$ Number of predicate nodes + 1
   $= 3 + 1$
   $= 4$
3. $V(G) =$ No. of regions
   $= 4$

Independent Paths of male_tax()

Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph, as shown below:

1. $N_1\ N_2\ N_6$
2. $N_1\ N_3\ N_6$
3. $N_1\ N_4\ N_6$
4. $N_1\ N_5\ N_6$

Test Case Design on male_tax() from the list of Independent Paths

Test Case ID	Input Taxable income	Expected Output	Independent path covered by Test Case
1	147000	return 0	1
2	123000	return 12300	2
3	340000	return 68000	3
4	620000	return 186000	4

Basis Path Testing of module fem_tax()

Source code of fem_tax()

```
float fem_tax(float taxable)
        {
        float t;

1       if(taxable<=180000)
2               {
3               return 0;
4               }
5       else if((taxable>180000)&&(taxable<=300000))
6               {
7               t=10*taxable/100;
8               }
9       else if((taxable>300000)&&(taxable<=500000))
10              {
11              t=20*taxable/100;
12              }
13      else
14              {
15              t=30*taxable/100;
16              }
17      return t;
18    }
```

DD Graph for Module fem_tax()

Cyclomatic Complexity of fem_tax()

1. $V(G) = e - n + 2P$

$$= 8 - 6 + 2$$

$$= 4$$

2. $V(G) = \text{Number of predicate nodes} + 1$
   $= 3 + 1$
   $= 4$
3. $V(G) = \text{No. of regions}$
   $= 4$

Independent Paths of fem_tax()

Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph, as shown below:

1. $N_1\ N_2\ N_6$
2. $N_1\ N_3\ N_6$
3. $N_1\ N_4\ N_6$
4. $N_1\ N_5\ N_6$

Test Case Design on fem_tax() from the list of Independent Paths

Test Case ID	Input Taxable income	Expected Output	Independent path covered by Test Case
1	180000	return 0	1
2	270000	return 27000	2
3	430000	return 86000	3
4	620000	return 186000	4

Basis Path Testing of module senior_tax()

Source Code of senior_tax()

```
float senior_tax(float taxable)
        {
        float t;

1       if(taxable<=225000)
2               {
3       return 0;
4               }
5       else if((taxable>225000)&&(taxable<=300000))
6               {
7               t=10*taxable/100;
8               }
9       else if((taxable>300000)&&(taxable<=500000))
10              {
```

```
11              t=20*taxable/100;
12              }
13      else
14              {
15              t=30*taxable/100;
16              }
17      return t;
18   }
```

DD Graph for Module senior_tax()

Cyclomatic Complexity of senior_tax()

1. $V(G) = e - n + 2P$
   $= 8 - 6 + 2$
   $= 4$
2. $V(G) =$ Number of predicate nodes + 1
   $= 3 + 1$
   $= 4$
3. $V(G) =$ No. of regions
   $= 4$

Independent Paths of senior_tax()

Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph as shown below:

1. $N_1\ N_2\ N_6$
2. $N_1\ N_3\ N_6$
3. $N_1\ N_4\ N_6$
4. $N_1\ N_5\ N_6$

Test case design on senior_tax from the list of independent paths

Test Case ID	Input Taxable income	Expected Output	Independent path covered by Test Case
1	225000	return 0	1
2	228000	return 22800	2
3	430000	return 86000	3
4	620000	return 186000	4

Basis Path Testing of Module gtd()

Source code of gtd()

```
void gtd(float gross_income,float net_deductions,float tax_ded)
        {
1       float taxable_income,tax,edu_cess,surcharge=0,difference,total_tax;
2       taxable_income=gross_income-net_deductions;

3       printf("\nTaxable Income\t\t:%f",taxable_income);
4       if((s=='m')||(s=='M'))
5               {
6               tax=male_tax(taxable_income);
7               printf("\nTax\t\t\t:%f",tax);
8               }
9       else if((s=='f')||(s=='F'))
10              {
11              tax=fem_tax(taxable_income);
12              printf("\n\Tax\t\t\t:%f",tax);
13              }
14      else if((2009-yr1)>65)
15              {
16              tax=senior_tax(taxable_income);
17              printf("\nTax\t\t\t:%f",tax);
18              }
19      if(tax>1000000)
20              {
21              surcharge=10*tax/100;
22              printf("\nSurcharge\t\t:%f",surcharge);
23              }
24      edu_cess=3*tax/100;
25      printf("\nEducational cess\t:%f",edu_cess);
26      total_tax=tax+surcharge+edu_cess;
27      printf("\nTax payable\t\t:%f",total_tax);
28      printf("\nTotal tax deducted\t:%f",tax_ded);
```

```
29      difference=total_tax-tax_ded;
30      if(difference>0)
31              {
32              printf("\nTax payable\t\t:%f",difference);
33              }
34      else if(difference<0)
35              {
36              printf("\nRerfundable\t%f",-1*difference);
37              }
38      else
39              {
40              printf("\nNo tax");
41              }
42   }
```

DD Graph for Module gtd()

Cyclomatic Complexity of gtd()

1. $V(G) = e - n + 2P$
   $= 17 - 13 + 2$
   $= 6$
2. $V(G) =$ Number of predicate nodes + 1
   $= 5 + 1$
   $= 6$
3. $V(G) =$ No. of regions
   $= 6$

Independent Paths of gtd()

Since the cyclomatic complexity of the graph is 6, there will be 6 independent paths in the graph as shown below:

1. $N_1\ N_2\ N_3\ N_6\ N_8\ N_9\ N_{10}\ N_{13}$
2. $N_1\ N_2\ N_4\ N_6\ N_8\ N_9\ N_{10}\ N_{13}$
3. $N_1\ N_2\ N_5\ N_6\ N_8\ N_9\ N_{10}\ N_{13}$
4. $N_1\ N_2\ N_3\ N_6\ N_7\ N_8\ N_9\ N_{10}\ N_{13}$
5. $N_1\ N_2\ N_3\ N_6\ N_7\ N_8\ N_9\ N_{11}\ N_{13}$
6. $N_1\ N_2\ N_3\ N_6\ N_7\ N_8\ N_9\ N_{12}\ N_{13}$

Test case design on gtd() from the list of independent paths

Test Case ID	Inputs					Expected Output	Independent path covered by Test Case
	Gross_ income	Net_ deductions	Tax_ded	Status	Age		
1	320000	2000	1000	male	34	Tax 30000 Educational cess 900 Tax payable 30900 Total Tax deducted 1000 Tax payable 29900	1
2	340000	2000	1000	female	34	Tax 32000 Educational cess 960 Tax payable 32960 Total Tax deducted 1000 Tax payable 31960	2
3	420000	2000	1000	male	67	Tax 84000 Educational cess 2520 Tax payable 86520 Total Tax deducted 1000 Tax payable 85520	1, 3

4	12000000	2000	1000	male	34	Tax 3629400 Surcharge 362940 Educational cess 108882 Tax payable 4101222 Total Tax deducted 1000 Tax payable 4100222	1, 4
5	320000	2000	32000	male	34	Tax 30000 Educational cess 900 Tax payable 30900 Total Tax deducted 32000 Refundable 1100	1, 5
6	320000	2000	30900	male	34	Tax 30000 Educational cess 900 Tax payable 30900 Total Tax deducted 30900 No tax	1, 6

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

References

1. Edward Kit, Software Testing in the Real World, Pearson Education Pte. Ltd., Delhi, 2004
2. G.J. Myers, The Art of Software Testing, John Wiley & Sons, 1979
3. Pankaj Jalote, CMM in Practice, Pearson Education Pte. Ltd., Delhi, 2004
4. William E. Perry, Effective Methods for Software Testing, John Wiley & Sons (Asia) Pte. Ltd., Second Edition, 2003
5. Boehm B., Software Engineering Economics, Prentice Hall, 1981
6. Wallace, D.R. and R.U. Fujii, "Software Verification and Validation: An Overview", IEEE Software, May 1989
7. Roger S. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill International Edition, Fifth Edition, 2001
8. K.K. Aggarwal, Yogesh Singh, Software Engineering, New Age International (P) Ltd., 2001
9. Boris Beizer, Software Testing Techniques, Dreamtech Press, Second Edition, 2004
10. J.M. Voas, "PIE: A Dynamic Failure based technique", IEEE Trans. On Software Engg., Vol. 18, No. 8, Aug. 1992, pp717-727
11. James A. Whittalcer, "What is Software Testing? And why is it so hard?", IEEE Software, Ja/Feb 2000, pp 70-79
12. Elfriede Dustin, Effective Software Testing: 50 specific ways to improve your testing, Pearson Education Pte. Ltd., Delhi, 2004
13. Renu Rajani, Pradeep Oak, Software Testing: Effective Methods, Tools and Techniques, Tata McGraw-Hill Publishing, 2004
14. Pankaj jalote, An Integrated approach to Software Engineering, Narosa Publishing House, Second Edition, 2003
15. Manna Z., and Waldinger R., "The Logic of Complete Programming", IEE Trans. On Software Engg., 4:199-229. 1978
16. Richard Fairley, Software Engineering Concepts, Tata McGraw-Hill, 2003
17. Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli, Fundamentals of Software Engineering, Prentice Hall of India, 1999
18. IEEE Recommended Practice for Software requirements Specifications IEEE Std 830-1998 (Revision of IEEE-Std 830-1993)
19. IEEE Recommended Practice for Software Design Descriptions, IEEE Std 1016-1998
20. Software Testing: A Craftsman's Approach, CRC Press, Second Edition, 2002
21. ANSI/IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, IEEE Computer Society Press
22. Louise Tamres, Introducing Software Testing, Pearson Education Pte. Ltd., Delhi, 2004
23. Kenneth H. Rosen, Discrete Mathematics and its applications, Tata McGraw-Hill, Forth Edition
24. T. McCabe, "A Complexity Measure", IEEE Trans. On Software Engg., 2(4), 1976, pp308-320
25. Fabrizio Riguzzi, "A Survey of Software Metrics", DEIS Tech. Report No. DEIS-LIA-96-010, July 1996
26. Basili V.R., Selby R.W., "Comparing the effectiveness of Software Testing Strategies", IEEE Trans. On Software Engg., SE-13(12), 1278-96, 1987
27. Gilb T., Graham D., Software Inspection, Addison Wesley, 1993

28. Fagan M.E., "Advances in Software Inspections", IEEE Trans. On Software Engg., SE-12(7), 744-51
29. Ian Sommerville, Software Engineering, Pearson Education Pte. Ltd., Delhi, Sixth Edition, 2004
30. N.H. Petschenik, "Practical Properties in Software Testing", IEEE Software 2(5):18-23
31. G.W. Jones, Software Engineering, Jon Wiley and Sons, 1990
32. Norman E. Fenton, Shari Lawrence Pfleeger, Software Metrics: A Rigorous and Practical Approach, Brooks/Cole, Second Edition
33. David Garmus, David Herron, Function Point Analysis: Measurement Practices for Successful software projects, Addison Wesley Information Tech. series
34. Paul Goodman, Practical Implementation of Software Metrics, McGraw-Hill, 1993
35. Ilene Burnstein, Practical Software Testing: A Process oriented approach, Springer-Verlog, New York, 2003
36. Grady Booch, Object –Oriented Analysis and Design with Applications, Pearson Education, Second Edition, 2000
37. Wendy Boggs, Michael Boggs, Mastering UML with Rational Rose, BPB Publications, 2002
38. M.J. Harrold, John D. McGregor and Kevin J. Fitzpatrick, "Incremental Testing of Object-Oriented Class Structures", Proceedings of 14th International Conference on Software Engg., 1992
39. P.B. Crisby, Quality is Free: The Art of making Quality Certain, McGraw-Hill, New York, 1979
40. J.M. Juran and F.M. Gryna, Quality Planning and Analysis: Product Development through use, McGraw-Hill, NewYork, 1970
41. W. Humphrey, A Discipline for Software Engineering, Addison Wesley, MA, 1995
42. Stephan H. Kan, Metrics and Model in Software Quality Engineering:, Pearson Education, 2003
43. M.E. Fagon, "Design and Code Inspections to reduce errors in program development" IBM System Journal, (3):182-211, 1976
44. I.S.G. of Software Engg. Terminology, IEEE Standards Collection, IEEE Std 610.12-1990, Sep. 1990
45. Akira K. Onoma, Wei-Tek Tsai, Mustafa Poonawal and Hiroshi Suganuma, "Regression Testing in an Industrial Environment", Communication of the ACM, 41(5):81-86, 1998
46. Hiralal Agrawal, Joseph R. Horgan et al., "Incremental Regression Testing", IEEE Conference on Software maintenance, Montreal Canal, 1993
47. H. Lenng and L. White, "Insights into Regression Testing", Proceedings of the conference on Software maintenance, 1989
48. Cem Kaner, Jack Falk, Hung Quoc Nguyen, Testing Computer Software, Second Edition, Wiley India, 2007
49. B. Beizer, Software Testing and Quality Assurance, Von Nostrand Reinhold, New York, 1984
50. Rothermel and M.J. Harrold, "A Framework for evaluating regression test selection techniques", Proceedings of 16th International Conference on Software Engineering, IEEE Computer Soceity Press, May 1994
51. Rothermel and M.J. Harrold, "Analyzing regression test selection techniques", IEEE Transactions on Software Engineering, 22(8):529-551, Aug. 1996
52. Rothermel and M.J. Harrold, "Empirical studies of a safe regression test selection technique", IEEE Transactions on Software Engineering, 24(6):401-419, June 1998
53. K. Fischer, F. Raji, A. Chruscicki, " A Methodology for retesting modified software", Proceedings of National Tele. conference , B-6-3, pp1-6, 1981
54. M. Harrold and M. Soffa, "An incremental approach to unit testing during maintenance" Proceedings of the conference on Software maintenance, pp 362-367, 1988
55. G. Rothermel and M. Harrold, "A safe, efficient regression test selection technique", ACM Transactions on Software Engg. And Methodology, 6(2):173-210, 1997
56. IEEE/ANSI (1983), IEEE Standard for Software Test Documentation, (Reaff. 1991), IEEE Std 829-1983
57. D.N. Card, R.L. Glass, Measuring Software Design Quality, Prentice-Hall, 1990
58. G. Rothermel, R.H. Untch, Chengyun Chu, M.J. Harrold, "Prioritizing test cases for Re-

gression Testing", IEEE Trans. On Software Engg., Vol. 27, Issue 10, Oct 2001

59. Paulk, M, Curtis B., Chrissis M.B. and Weber, C., *Capability Maturity Model for Software*, Software Engineering Institute, Carnegie Mellon University, 1993
60. Burnstein I., Suwanassart T., and Carlson C., "Developing a Testing Maturity Model. Part 1", *Crosstalk, Journal of Defense Software Engineering*, 9, no. 8, 21–24, 1996, also available on http://www.stsc.hill.af.mil/-crosstalk/1996/aug/developi.asp
61. Burnstein I., Suwanassart T., and CarlsonC., "Developing a Testing Maturity Model. Part 2", *Crosstalk, Journal of DefenseSoftware Engineering*, 9, no. 9, 19–26, 1996, also available on http://www.stsc.hill.af.mil/-crosstalk/1996/sep/developi.asp
62. Gelperin D. and Hetzel B., "The Growth of Software Testing", Communications of the ACM, 31, no. 6, 687–695, 1998
63. Paulk, M., C. Weber, B. Curtis, and M. Chrissis, *The Capability Maturity Model Guideline for Improving the Software Process*, Addison-Wesley, Reading, Mass., 1995
64. Thayer, R., "Software Engineering Project Management, A Top Down View," *IEEE Tutorial, Software Engineering Project Management,* R. Thayer, ed., IEEE Computer Society Press, Los Alamitos, Calif., 1990, pp. 15-53
65. Jef Jacobs, Jan van Moll, and Tom Stokes, "The Process of Test Process Improvement", XOOTIC MAGAZINE, Nov. 2000
66. S.L. Pfleeger, Software Engineering:Theory and Practice, Pearson Education, Second Edition, 2003
67. Firesmith, Donald G. "Testing Object-Oriented Software," published in Proceedings of TOOLS, 19 March 1993
68. David et al. "Developing an Object-Oriented Software Testing Environment", Communications of the ACM, Vol. 38, No. 10, pp. 75-87, Oct. 1995
69. Booch, G., *Object Oriented Design With Applications*, Benjamin Cummings, 1991,
70. Harrison, R., *Abstract Data Types in Modula-2*, J. Wiley & Sons, 1989
71. C. D. Turner and D. J. Robson, "Guidance for the Testing of Object-Oriented Programs", Technical Report No: TR 2/93 Computer Science Division School of Engineering and Computer Science (SECS) University of Durham Durham, England16 April, 1993
72. T. J. Ostrand and M. J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," Comm. ACM, vol. 31, no. 6, pp. 676–686, 1988.
73. P. Wegner and S. B. Zdonik, "Inheritance as an incremental modification mechanism or what like is and isn't like," Proceedings of ECOOP'88, pp. 55-77, Springer-Verlag, 1988
74. Arthur H.W., Thomas J. McCabe, "Structured Testing: A Testing Methodology using the Cyclomatic Complexity metric", NIST special publication 500-235, sep. 1996
75. P. Jorgensen and C. Erickson, Object-oriented integration testing, *Communications of the ACM*, 37(9):30-38, Septenber 1994
76. J. Overbeck, Integration Testing for Object Oriented Software, PhD thesis, Vienna University of Technology, 1994.
77. S. Kirani, Specification and Verification of Object-Oriented Programs, PhD thesis, University of Minnesota, Minneapolis, Minnesota, December 1994
78. P. Jorgensen and C. Erickson, "Object-oriented integration testing", Communicationsof the ACM, 37(9):30.38, September 1994
79. D. Gelperin and B. Hetzel, "The Growth of Software Testing", Communications of the ACM, Volume 31 Issue 6, June 1988, pp. 687-695
80. Lu Luo, "Software Testing Techniques: Technology Maturation and Research Strategies", Institute for Software Research International, Class Report for 17-939A
81. Software Testing 3.0:The Continuing Evolution of Software Testing, LogiGear Corporation white paper January 2008
82. Bret Pettichord, Schools of Software Testing, March 2007, www.pettichord.com
83. James Bach, Explaining testing anybody, www.satisfice.com
84. Edward Miller, "Introduction to software testing technology", In Tutorial: Software Testing & Validation Techniques, pages 4-16, IEEE Computer Society Press, 1981.
85. Cem Kaner, "Exploratory Testing", Quality Assurance Institute Worldwide Annual Software

Testing Conference, Orlando, FL, November 2006

86. l.H. Fenton, "Response to the SHARE software service task force report, IBM Corp., Kingston, NY, March 6, 1984
87. Parnas, D. L. and Weiss, D. M., "Active Design Reviews: Principles and Practices," Proceedings of the Eighth International Conference on Software Engineering, August, 1985
88. Johnson, P. M., "An Instrumented Approach to Improving Software Quality Through Formal Technical Review", Proceedings of the 16th International Conference on Software Engineering (ICSE-16), May 1994
89. Gilb, T. and Graham, D., "Software Inspection", Addison-Wesley, 1993
90. Macdonald, F. and Miller, J., "Modeling Software Inspection Methods for the Application of Tool Support", December, 1995
91. Martin, J. and Tsai, W. T., "N-Fold Inspection: A Requirements Analysis Technique", Communications of the ACM, V. 33, N. 2, February, 1990, pp. 225-232
92. Knight, J. C., and Myers, E. A., "An Improved Inspection Technique", Communications of the ACM, V. 36, N. 11, November, 1993, pp. 51-61
93. Porter, A. & Votta, L.G., "An experiment to assess different defect detection methods for software requirements inspections", In Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, Los Alamitos: IEEE Computer Society Press, 103-112, May 16-21, 1994
94. Basili, V.R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Soerumgaard, S. & Zelkowitz M., "The empirical investigation of perspective-based reading", Empirical Software Engineering 1(2), 133-164, 1996
95. Laitenberger, O. & DeBaud, J.-M., "Perspective-based reading of code documents at Robert Bosch GmbH", Information and Software Technology 39(11), 781-791, 1997
96. Thelin, T., Runeson, P. & Regnell, B., "Usage-based reading - An experiment to guide reviewers with use cases" Journal of Information and Software Technology 43(15), 925-938, 2001
97. Thelin, T., Runeson, P. & Wohlin, C., "An experimental comparison of usage-based and checklist-based reading", IEEE Transactions on Software Engineering 29(8), 687-704, 2003
98. Thelin, T., Runeson, P., Wohlin, C., Olsson, T. & Andersson, C., "Evaluation of usage-based reading - Conclusions after three experiments", Empirical Software Engineering 9(1-2), 77-110, 2004
99. Dunsmore A., Roper M. & Wood M., "Systematic object-oriented inspection technique", In Proceedings of the 23rd International Conference on Software Engineering, Toronto, Canada, May 12-19. Washington: IEEE Computer Society, 123-144, 2001
100. Dunsmore A., Roper M. & Wood M., "Further investigations into the development and evaluation of reading techniques for object-oriented code inspection", In Proceedings of the 24th International Conference on Software Engineering, Orlando, Florida, May 19-25. New York: ACM Press, 47-57, 2002
101. Dunsmore, A., Roper, M. & Wood, M., "The development and evaluation of three diverse techniques for object-oriented code inspection", IEEE Transactions on Software Engineering 29(8), 677-686, 2003
102. Kelly, D. & Shepard, T., "Task-directed software inspection", Journal of Systems and Software 73(2), 361-368, 2004
103. Cheng, B. and Jeffrey, R., "Comparing Inspection Strategies for Software Requirements Specifications", Proceedings of the 1996 Australian Software Engineering Conference, pages 203-211, 1996
104. Basili, V., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sorumgard, S., and Zelkowitz, M., "The Empirical Investigation of Perspective-based Reading", Journal of Empirical Software Engineering, 2(1):133-164, 1996
105. Laitenberger, O., "Cost-Effective Detection of Software Defects with Perspective-based Inspection", PhD-Thesis, University of Kaiserslautern, ISBN 3-8167-5583-6, 2000
106. Edward Yourdon, Structured Walkthroughs, Yourdon Press, 4th edition, 1989
107. Laitenberger, O., "Studying the effects of code inspection and structural testing on software quality", Proceedings of ninth International Symposium o Software Relibility Engineering, 1998

108. W. Humphrey, A Discipline for Software Engineering, Addison-Wesley, Reading, MA, 1995
109. J. Rakos, Software Project Management for small to medium-sized projects, Prentice Hall, Englewood Cliffs, NJ, 1990
110. Y. Chernak, "Validating and improving test case effectiveness", IEEE Software, Vol. 16, No.1, pp. 81-86, 2001
111. Drs Eric P W M Van Veenendaal CISA, Ton Dekkers, "Test Point Analysis: A Method for Test Estimation", Published in Project Control for Software Quality, Shaker Publishing BV, Maastricht, Netherlands, 1999
112. S. R. Rakitin. Software Verification and Validation for Practitioners and Managers. Artech House Inc. Boston-London, 2001.
113. QAI Consulting Organization. Emphasizing Software Test Process Improvement, http://www.qaiindia.com/Resources_Art/journal_emphasizing.htm , September 2006.
114. M. L. Hutcheson. Software Testing Fundamentals: Methods and Metrics. John Willey & Sons, 2003
115. K. Iberle, S. Bartlett. Estimating Tester to Developer Ratios (or Not). Hewlett-Packard and STEP Technology, www.kiberle.com/pnsqc1/estimate.doc, November 2006
116. R. S. Pressman. Software Engineering – A Practitioner's Approach. Sixth Edition, McGraw Hill Education Asia, 2005.
117. R. D. Craig, S. P. Jaskiel. Systematic Software Testing. Artech House Publishers, Boston-London, 2002.
118. B. Berger. Evaluating Test Plans with Rubrics. International Conference on Software Testing Analysis and Review, 2004
119. P. Piwowarski, M. Ohba, J. Caruso. Coverage Measurement Experience During Function Test. International Business Machines Corporation. In IEEE Software Engineering Proceedings, 1993.
120. M. Marre´, A. Bertolino. Using Spanning Sets for Coverage Testing. In IEEE Transactions on Software Engineering, Volume 29, Number 11, November 2003.
121. Afzal, W., Torkar, R., "Incorporating Metrics in an Organizational Test Strategy", Proc of IEEE International Conference on Software Testing Verification and Validation Workshop, April 2008
122. David Garmus, David Herron, Function Point Analysis: Measurement Practices for Successful Software Projects, Addison Wesley, 2001
123. www.ifpug.org
124. R. S. Pressman. Software Engineering – A Practitioner's Approach. McGraw Hill Education Asia, 2005.
125. W. Dijkstra. Structured Programming. In J.N.Buxton and B.Randell (eds.), Software Engineering Techniques, Brussels, Belgium, NATO Science Committee, 1970
126. E. Miller. The Philosophy of Testing. In Program Testing Techniques, IEEE Computer Society Press, 1977
127. M. Shaw, "Abstraction techniques in Modern Programming Languages", IEEE Software, vol. 1(4), 1984
128. J.Musa, Software Reliability, New York, NY: McGraw-Hill, 1999
129. J.Musa, "Software-Reliability-Engineered Testing", IEEE Computer, vol. 29, pp. 61-68, Nov. 1996
130. H. Agarwal, J.R. Horgan, E.W. Krauser, S. London, "Incremental regression testing", IEEE International Conference on Software Maintenance, pp. 348-357, 1993
131. T. Gyimothy, A. Beszedes, I. Forgacs, "An efficient relevant slicing method for debugging", ACM/SIGSOFT Foundations of Software Engineering, pp. 303-321, 1999
132. B. Korel, J.Laski, "Algorithmic software fault localization", Annual Hawaii International conference on system sciences", pp. 246-252, 1991
133. S. Elbaum, A. Malishevsky, G. Rothermel, "Prioritizing test cases for regression testing", Proceedings of International symposium on software testing and analysis, pp. 102-112, Aug. 2000
134. G. Rothermel, R. Untch, C. Chu, M.J. Harrold, "Test case Prioritization: an empirical study", Proccedings of International conference on software maintenance, pp. 179-188, Aug. 1999
135. S. Elbaum, A Malishevsky, G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization", Proceedings of 23rd International conference on software engineering, Ontario, Canada, May 2001

136. H. Srikanth, L. Williams, J. Osborne, "Towards the Prioritization of system test cases" North Carolina State University TR-2005-44, 2005
137. R.A. DeMillo, R.J. Lipton, F.G. Sayward, "Hints on Test Data Selection: Help for the practicing programmer", Computer, 11(4):34-41, April 1978
138. R.G Hamlet, "Testing programs with the aid of a compiler", IEEE transactions on Software engineering, SE-3(4):279-290, July 1977
139. Dennis Jeffrey, Neelam Gupta, "Test Case Prioritization using Relevant Slices", Proceedings of 30th Annual International Computer software and applications conference (COMPSAC '06), vol. 1, pp. 411-420, 2006
140. Koomen, T. and Pol M., Test Process Improvement: A practical step-by step guide to structured testing, ACM Press, London, England, 1999
141. Sog04 TPI home pages, Sogeti Nederland B.V., 2004. http://www.sogeti.nl/index.html?/iospagina.cfm?uNr=150
142. TMa04 TMap home pages, TMap - Sogeti Nederland B.V., 2004, http://www.tmap.net
143. www.gerrardconsulting.com
144. Ericson, T., Subotic, A. and Ursing, S., "Towards a Test Improvement Model", Proceedingsof the Fourth European Conference on Software Testing, Analysis & Review, Amsterdam, December 2-6, 1996
145. Jari Andersin, "TPI – a model for Test Process Improvement", Seminar on Quality Models for Software Engineering, Department of Computer Science UNIVERSITY OF HELSINKI, Helsinki, 5th October 2004
146. Capers, Jones, Applied software measurement, McGraw-Hill, 1996
147. Ghulam Mustafa, Abad Ali Shah, et al, "A Strategy for testing of web based software", Information Technology Journal, 6(1):74-81, 2007
148. Dart, S., Containing the Web Crisis Using Configuration Management, Proc ICSE Workshop on WebEngineering, 1999. http://fistserv.macarthur.uws.edu.au/san/icse99-webe/
149. Murugesan, S. et al. "Web engineering: A New Discipline for Development of Web based systems" In Proceedings of the First ICSE Workshop on Web Engineering, Los Angeles (pp. 1-9), 1999
150. Deshpande, Y. et al., " Web engineering", Journal of Web Engineering, 1(1), 3-17, 2002
151. Deshpande, Y., Ginige, A., Murugesan, S., & Hansen, S., "Consolidating Web engineering as a discipline.", SEA Software, (April), 32-34, 2002
152. Deshpande, Y., & Hansen, S., "Web engineering: creating a discipline among discipline", IEEE Multimedia, (April - June), 82-87, 2001
153. Deshpande Y., Olsina, L., & Murugesan, S., "Web engineering.", Report on the Third ICSE Workshop on Web Engineering, ICSE2002, Orlando, FL, USA, 2002.
154. Ginige, A., & Murugesan, S., "Web engineering: An introduction", IEEE Multimedia, 8(1), 14-18, 2001
155. Ginige, A. & Murugesan, S., "The essence of Web engineering: Managing the diversity and complexity of Web application development", IEEE Multimedia, 8(2), 22-25, 2001
156. Ginige, A., & Murugesan, S., "Web engineering: A methodology for developing scalable, maintainable Web applications", Cutter IT Journal, 14(7), 24-35, 2001
157. HENNICKER R. and KOCH N., "A UML-based Methodology for Hypermedia Design", In Proceedings of the Unified Modeling Language Conference, UML´2000, Evans A. and Kent S., Eds. LNCS 1939, Springer Verlag, 410-424, 2000
158. KOCH N. , Software Engineering for Adaptive Hypermedia Applications, PhD. Thesis, Reihe Softwaretechnik 12, Uni-Druck Publishing Company, Munich, 2001
159. KOCH N., KRAUS A. and HENNICKER R., "The Authoring Process of the UML-based Web Engineering Approach", In First International Workshop on Web-Oriented Software Technology IWWOST'2001, Valencia, 2001
160. KRAUS A. and KOCH N. , "Generation of Web Applications from UML Models using an XML Publishing Framework", Proceeding of the Integrated Design and Process Technology Conference, IDPT'2002, Pasadena, 2001
161. Koch, Andreas Kraus, "The Expressive Power of UML-based Web Engineering", Ludwig-Maximilians-Universität München. Germany http://www.pst.informatik.uni-muenchen.de/personen/kochn/IWWOST02-koch-kraus.pdf

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

APPENDICES

Appendix

A

Answers to Multiple Choice Questions

Chapter 1

1. (b)	2. (a)	3. (c)	4. (d)	5. (b)
6. (b)	7. (a)	8. (a)	9. (b)	10. (c)

Chapter 2

1. (b)	2. (a)	3. (d)	4. (d)	5. (c)
6. (a)	7. (c)	8. (d)	9. (b)	10. (a)
11. (b)	12. (a)	13. (c)	14. (b)	15. (b)

Chapter 3

1. (c)	2. (c)	3. (a)	4. (b)	5. (c)
6. (d)	7. (b)	8. (d)	9. (d)	10. (d)
11. (a) and (c)	12. (c)	13. (b)		

Chapter 4

1. (b)	2. (a)	3. (d)	4. (b)	5. (c)
6. (b)	7. (a)	8. (c)	9. (d)	10. (c)
11. (a)				

Chapter 5

1. (b)	2. (b)	3. (a)	4. (b)	5. (c)
6. (d)	7. (c)	8. (c)	9. (b)	10. (c)
11. (b)	12. (a)	13. (b)	14. (a)	15. (d)

Chapter 6

1. (a)	2. (b)	3. (b)	4. (d)	5. (a)
6. (b)	7. (d)	8. (d)	9. (b)	10. (a)
11. (a) and (c)	12. (b)	13. (c)	14. (b)	15. (d)
16. (a)	17. (c)	18. (c) and (d)	19. (d)	20. (d)

Chapter 7

1. (b)	2. (a)	3. (c)	4. (c)	5. (d)
6. (a)	7. (c)	8. (a)	9. (a)	10. (b)
11. (d)	12. (c)	13. (a)	14. (a) and (c)	15. (b)

16. (b)	17. (c)	18. (c)	19. (d)	20. (d)
21. (b)	22. (a)	23. (c)	24. (c)	

Chapter 8

1. (b)	2. (a)	3. (b)	4. (c)	5. (b)
6. (a)	7. (b)	8. (a) & (b)	9. (c)	10. (a)
11. (b)	12. (d)	13. (c)	14. (a)	15. (b)
16. (d)				

Chapter 9

1. (a), (c) and (d)	2. (b)	3. (c)	4. (a) and (b)	5. (a)
6. (c)	7. (b)	8. (a)	9. (b)	10. (c)
11. (d)	12. (a)	13. (d)	14. (a) and (b)	

Chapter 10

1. (a)	2. (a)	3. (a)	4. (b)	5. (c)
6. (b)	7. (a)			

Chapter 11

1. (c)	2. (b)	3. (a)	4. (b)	5. (c)
6. (a)	7. (a)	8. (b)	9. (c)	10. (d)
11. (a)	12. (b)			

Chapter 12

1. (a)	2. (b)	3. (a)	4. (b)	5. (a)
6. (b)	7. (a)	8. (c)	9. (c)	

Chapter 13

1. (d)	2. (b)	3. (a)	4. (b)	5. (a)
6. (c)	7. (a)	8. (d)	9. (a)	10. (b)
11. (c)	12. (a)	13. (b)		

Chapter 14

1. (b)	2. (d)	3. (a)	4. (c)	5. (d)
6. (b)	7. (b)	8. (b)	9. (d)	10. (c)
11. (a)				

Chapter 15

1. (a)	2. (c)	3. (a)	4. (d)	5. (c)
6. (b)	7. (c)	8. (a)	9. (b)	

Chapter 16

1. (b)	2. (a)	3. (c)	4. (a)	5. (d)
6. (c)	7. (a)	8. (b)	9. (b)	10. (a)
11. (c)	12. (d)	13. (d)	14. (d)	15. (c)
16. (a)	7. (b)	18. (a) and (d)	19. (b)	20. (b)

Chapter 17

1. (b)	2. (c)	3. (a)	4. (a)	5. (b)
6. (a)	7. (b)	8. (d)	9. (a)	10. (a)
11. (b)	12. (c)	13. (a)	14. (b)	15. (a) and (b)

Chapter 18

1. (b)	2. (b)	3. (a)	4. (b)	5. (d)
6. (c)	7. (a)			

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Appendix

B

Software Requirement Specification (SRS) Verification Checklist

S.No.		Y/N/NA	Remarks
1.	**Overview and System Functional Flow**		
	Is the overview of high-level system mentioned?		
	Is the system's functional flow clearly and completely described?		
	Has the high-level functionality of system depicted in a diagram like DFD level 0?		
	Do the high-level diagrams depict internal and external interfaces and data flows?		
	Are the software functions mentioned at a high-level with the viewpoint of operational system?		
	Are the users of the software including other systems interacting with it recognized?		
2.	**Functional Requirements**		
	Are the requirements traceable to customer?		
	Are the requirements defined separately?		
	Are the functionalities given proper name and unique number/ID?		
	Does each function fully define its purpose and scope?		
	Does each function mention required inputs and outputs?		
	Do the requirements specify any method for exception handling or handling any alternate functional logic flow?		
	Does each functional requirement mention the acceptance criteria, if any?		
	Have all the constraints (software / hardware) related to any functional requirement mentioned?		
	Are the functional requirements prioritized and given an identifier to indicate its importance?		
3.	**Non-functional (Performance, security, quality) requirements**		
	Is the acceptable response time for each functionality mentioned?		
	Is the number of maximum users on the system mentioned?		
	Is the number of transactions / online transactions per user per unit time mentioned?		

S.No.		Y/N/NA	Remarks
	Is the peak data flow volume mentioned?		
	Is the maximum size of the data transaction specified?		
	Is the upper and lower control limits (tolerance criteria) specified?		
	Is it mentioned what will be done by the system when it fails on exceeding its capacity?		
	Are all the physical and operational security and safety requirements, if any, mentioned?		
	Are quality requirements like reliability, portability, maintainability, etc. mentioned?		
	Are quality requirements quantified with the acceptable limit?		
	Are the non-functional requirements prioritized and given an identifier to indicate its importance?		
4.	**Interface Requirements**		
	Are all inputs to the system specified, including their source, accuracy, range of values, and parameters?		
	Are all screen formats specified?		
	Are all report formats specified?		
	Are all interface requirements between hardware, software, personnel, and procedures included?		
	Are all communication interfaces specified?		
5.	**Resource Requirements**		
	Are all hardware configuration requirements mentioned?		
	Are all software configuration requirements mentioned?		
	Are all memory requirements mentioned?		
	Are all software, hardware, memory requirements for interfaced systems mentioned?		
6.	**Correctness**		
	Is each requirement in scope for the project?		
	Is each requirement free from content and grammatical errors?		
	Does the requirement make the clear understanding?		
	Is the requirement realistic under mentioned constraints?		
7.	**Ambiguity**		
	Has every requirement only one interpretation?		
	Is each characteristic of the final product described using a single unique term?		
	Is any requirement conflict with other requirement?		
8.	**Consistency**		
	Is there any real world objects conflict such that as one specification recommends mouse for input and another recommends ball tab?		
	Is there any logical conflict between two specified actions?		

S.No.		Y/N/NA	Remarks
	Are there any requirements that describe two or more actions that conflict temporally?		
	Are there any requirements describing the same object that conflict with other requirements with respect to terminology?		
	Is there any conflict between the specified requirement and project goals?		
	Are the timing and memory limits compatible with hardware constraints?		
9.	**Completeness**		
	Is there any missing requirement?		
	Are requirements in terms of input, output, functional, non-functional, interfaces complete?		
	Is there any additional predicted capability for the system?		
	Are all requirements written at an appropriate level of detail?		
	Are assumptions and dependencies explicitly mentioned?		
	Are all sections, figures and tables labeled appropriately?		
	Are all figures and tables referenced appropriately?		
	Has every acronym, constant, variable, etc. been defined in the data dictionary?		
	Is a dictionary for all data elements provided?		
	Is the data dictionary complete in all manners?		
	Are all documents to be referenced listed?		
	Are all units of measure needed defined?		
10.	**Traceability**		
	Is each functional requirement traceable to system functionality?		
	Has each requirement a unique name or reference number in all the documents?		
11.	**Standards Compliance**		
	Is established standard or guidelines mentioned to be followed for the document?		
	Does the document format conform to the specified standard/guideline?		
	Are the standards and naming conventions followed according to the standards throughout the document?		

Y = Yes, N = No, NA = Not applicable

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Appendix

C

High Level Design (HLD) Verification Checklist

S.No.		Y/N/NA	Remarks
1.	**Data Design**		
	Have the sizes of data structure been estimated appropriately?		
	Is there any provision of overflow in a data structure?		
	Are the data formats consistent with the requirements?		
	Is the data usage consistent with its declaration?		
	Are the relationships correct among data objects in data dictionary?		
	Are databases and data warehouses consistent with requirements in SRS?		
	Are the data structure names meaningful?		
	Is the database organization and content specified?		
	Is the data model designed correct in all manners?		
	Are the database design rules listed?		
	Has the database model been described?		
	Are the logical data models of database, if any written clearly?		
2.	**Architectural Design**		
	Does the architecture consider every functional and non-functional requirements specified in SRS?		
	Is the architecture correct and unambiguous?		
	Is the functionality of sub-system traceable to the system functionality in SRS?		
	Is the hierarchy of subsystems correct?		
	Is dependency and interfaces between subsystems correctly designed?		
	Does the design consider all the constraints mentioned in SRS?		
	Is the coupling between subsystems low?		
	Is the cohesion between subsystems high?		
	Is the design flexible enough for future extensions to the program?		
	Does the architecture consider all exception handling features?		
	Are all input-outputs for a subsystem correct and complete?		
3.	**Interface Design**		
	Have the possible interfaces been identified?		

S.No.		Y/N/NA	Remarks
	Are the interfaces clear and well-defined?		
	Are the interfaces between modules according the to architecture design?		
	Is the required data passed at each interface?		
	Are the interfaces between software and other non-human producer and consumer of information correct?		
	Are the interfaces between the user and software system correct?		
	Are the interfaces consistent?		
	Are the response time for all the interfaces within required ranges?		
	Is the representation of Help in its desired manner?		
	Does the user return to the normal interaction from Help?		
	Do the error messages clarify the problem?		
	Does the message provide constructive advice for recovering from the error?		
	Is the mapping between every menu option and corresponding command for typed command interaction correct?		

Y = Yes, N = No, NA = Not applicable

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Appendix

D

Low Level design (LLD) Verification Checklist

S.No.	Criteria	Y/N/NA	Remarks
1.	Is HLD traceable to LLD?		
2.	Is high level and low level abstraction consistent?		
3.	Is every subsystem detailed enough to be operational?		
4.	Does the pseudo-code of every subsystem follow the guidelines and syntax rules of a program design language (PDL)?		
5.	Does the algorithm of every subsystem consistent with its defined functionality?		
6.	Does the design notation for LLD support the development of modular software?		
7.	Does the design notation for LLD support the means for interface specification?		
8.	Is the design notation able to represent local and global data?		
9.	Is the design notation able to modify the design representation easily?		
10.	Is the logic in each algorithm clear, correct, and complete?		
11.	Has the data been properly defined and initialized?		
12.	Are all the defined data used somewhere?		
13.	Is every detail of data available?		
14.	Is the data used consistently throughout the module and module interface?		
15.	Are any variables missing?		
16.	Have all interfaces been correctly considered and implemented in LLD?		
17.	Do the subsystems accept all data within the allowable range?		
18.	Is the data converted according to the correct format?		
19.	Has any consideration been given to the effects of round-off or truncation?		
20.	Are the indices valid in the subsystem pesudocode?		
21.	Is there any infinite loop in the subsystem pesudocode?		
22.	Does the design address arithmetic overflow and underflow?		
23.	Are the physical data models of database, if any, written clearly?		

Y = Yes, N = No, NA = Not applicable

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Appendix

E

General Software Design Document (SDD) Verification Checklist

S.No.	Criteria	Y/N/NA	Remarks
1.	Does the design allow addition of more parameters, for example, number of users?		
2.	Does the system allow addition of new data types?		
3.	What is the impact of adding new databases?		
4.	Can the design cope with technological changes?		
5.	Does every design decision documented in SDD have only a single interpretation?		
6.	Is the SDD consistent with higher-level documents?		
7.	Is the SDD consistent with documents and models at the same level?		
8.	Is the SDD internally consistent in that all design decisions that it contains are compatible?		
9.	Have all the design standards been followed?		
10.	Does SDD specify all significant design decisions?		
11.	Is the Architecture design supported with a diagram?		
12.	Have the actual deployment environment of the architecture in terms of tools, databases, and actual software used been mentioned?		

Y = Yes, N = No, NA = Not applicable

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Appendix

F

Generic Code Verification Checklist

S.No.		Y/N/NA	Remarks
1.	**Data Declaration**		
	Are all data considered in design declared in coding?		
	Are variable type and dimensions correctly declared?		
	Do all variables have proper type consistency?		
	Are declared variables initialized also? If so, are they correctly defined?		
	Are there any variables having same or similar name with any other library function or reserved keywords?		
	Is there any variable declared, but not used?		
	Are all the pointer variables declared correctly?		
	Are all the global variables declared correctly?		
	Is Boolean variable declared correctly (if supported)?		
	Is Boolean variable declared directly (if not supported by language directly)?		
	Are the variables of user-defined type declared correctly?		
2.	**Data Reference**		
	Is there any variable referenced, but not declared?		
	Is there any variable referenced, but not initialized?		
	Are the subscript values for array references within the specified limits?		
	Are the subscript values for array references of the type integer?		
	If a variable is passed as parameter in procedure, is it correctly and consistently referred?		
	Are the pointer variables correctly referenced?		
	Is Boolean variable referenced correctly?		
	Are the variables of user-defined data type referenced correctly?		
3.	**Interfaces**		
	Is the order of parameters same in prototype, calling module, and called module?		
	Do the types of parameters match in prototype, calling module, and called module?		

S.No.		Y/N/NA	Remarks
	Do the sizes of parameters match in prototype, calling module, and called module?		
	Does the return type of module match in prototype, calling module, and called module?		
	Does the called module after computation return the same type of data as expected?		
	Are the interfaces being implemented traceable to SDD and SRS?		
	Are the global variables referenced consistently?		
	Are the library files included correctly?		
	Are the called modules return to the calling module?		
4.	**Computation**		
	Are the data types of variables used in computation inconsistent?		
	Are the lengths of variables used in computation same?		
	Is the data type on both sides of computation same?		
	Is there any computation having divisor as zero?		
	Are the order of evaluation in computation correct?		
	Are the order of precedence of operators correct?		
	Are mixed-mode computations correct?		
	Is loop index variable used correctly?		
	Are bitwise operators used correctly in the computation?		
	Are logical operators used correctly in the computation?		
	If there is a computation such that on the right side, there is a module call and the result returned is stored in the variable on the left side, then is the data type of variable on the left side of computation same as of the data type of value returned by the module call?		
	Are special operators used correctly in the computation?		
5.	**Comparison**		
	Are comparison operators used correctly?		
	Do the types of variables, for which comparison is made, match?		
	Are logical expressions expressed correctly?		
	Are the operands of Boolean operator Boolean?		
6.	**Control Flow**		
	Is there any infinite loop?		
	Is there any loop which does not enter in its body?		
	Is there any loop which does not by-pass?		
	Is there any if-then-else/switch-case structure infinite?		
	Is there any if-then-else/switch-case structure which does not enter in its body?		
	Is there any if-then-else structure which does not by-pass?		
	Are the nested control statements used correctly?		
	How many nested controls are there?		

S.No.		Y/N/NA	Remarks
	Is the expression blank in a loop?		
	Is any required expression missing in a loop?		
	Is each loop controlled by a different index?		
	Is one loop completely embedded within the other in case of nested loops?		
	Is the expression missing in an if-then-else structure?		
	Is the type of expression in switch-case structure an integer?		
	Are all Case labels in switch-case structure unique?		
	Does the control come out from each Case label group?		
	Is there any 'goto' statement?		
	Does every switch-case contain a default label?		
7.	**Input/Output**		
	Are the input statements according to the specified format of language?		
	Are the specified input ranges correct in input statement format?		
	Are the output statements according to the specified format of language?		
	Are the specified output ranges to be displayed as output correct in output statement format?		
	Are the files declared correct?		
	Are the files opened before use?		
	Are the parameters in File Open specified correctly?		
	Are all opened files closed?		
	Are End of File conditions detected?		
	Are the texts displayed to user on the screen are meaningful and appropriate?		
	Is there any spelling or grammatical mistake in the texts displayed to the user on the screen?		
8.	**Modules/Sub-systems**		
	Is there any traceability between LLD and module code?		
	Are all modules coded which were considered in LLD?		
	Is there any module missing which was considered in LLD?		
	Does the module's code match with the pseudo-code in the LLD?		
	Is there any logical mistake in the module code which contradicts its functional specification?		
	Is there any module which has not been referenced?		
	Is the implementation of module according to the language constraints complex? If so, can it be restructured with modifications in pseudo-code?		
9.	**General**		
	Are there comments put appropriately in the code?		
	Are the comments put for complex logic of the code?		
	Are the comments meaningful and understandable?		
	Do all source code files mention the function number corresponding to which it has been developed?		

S.No.		Y/N/NA	Remarks
	Are there proper indentations in the code?		
	Is there an 'end' brace for every 'begin' in all structures?		
	Is the code readable enough?		
	Can the compact code be replaced with the alternate code which is more readable and understandable?		
	Does the code adhere to coding standards?		
	Are there any leftover stubs or test drivers in the code?		
	Is the code consistent in style?		
	Is there any unreachable code?		
	Is there any redundant code which can be replaced by the component library module?		
	Have all the declared storage been used to their full limit or is there any storage space wastage?		
	Is there any inconsistency in code and comments?		

Y = Yes, N = No, NA = Not applicable

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

Acknowledgements

I am thankful to God for making things possible at the right time always.

Big projects are not developed overnight. Some ideas always incubate in our subconscious and take a definite shape gradually. However, these ideas are not developed by their own; they take shape as a result of constant learning and interaction between individuals and great personalities. I would like to acknowledge these personalities who have inspired me directly or indirectly to work on this book.

I would like to express my sincere gratitude to my school-time teacher, Sh. Girish Kumar who had given me the base for my upcoming life at that time. He has always been a role model for me. Next, I would like to thank my Guru, Pandit Priyadutt Shastri for realizing life with a totally different viewpoint and caused a turning point in my life. The principles that I learnt from these two persons will always be a moral support for me in any project.

The technical roots behind writing this book date back to the days when I was working in the Central Research Laboratory (Bharat Electronics Ltd., Ghaziabad), where I learnt many practical techniques of software testing. But for the critical learning support of Sh. K. Johri (Scientist at Central Research Laboratory, Ghaziabad) I would not have learnt this discipline. Those learning have helped me a lot in writing this book. He always used to say, 'Welcome the bugs; do not hide them.' While explaining the psychology of software testing, I kept his saying in mind

I would like to express my profound gratitude to Dr A. K. Sharma, Chairman (Computer Engg.), YMCA University of Science and Technology, Faridabad, who showed me the path of research and technical writing during the research work performed under him. I am extremely grateful to all my colleagues with whom I discussed many issues. Many thanks to my students, Sandeep Rana, Anita, Harsh, InduBala, and all others for their contribution towards completing this book.

I am indebted to my family for their love, encouragement, and support throughout my education. I am also thankful for all the support received from my parents-in-law.

I owe a lot to my dear wife, Anushree, who made many compromises to let me finish this book. I am thankful for her never-ending patience, unconditional moral support, and peaceful environment at home. Without her friendship and love, this book would not have been completed. I express my heartfelt gratitude to my dear daughter, Smiti, for her love and encouragement.

Finally, I extend my gratitude to the editorial staff at Oxford University Press for their support.

Thanks to all of you!

Naresh Chauhan

Copyrighted Materials
Copyright © 2010 Oxford University Press Retrieved from www.knovel.com

INDEX

Index Terms	Links		
A			
abstraction	447		
abstract superclass	460		
acceptance criteria	291		
acceptance testing	81	197	245
alpha testing	244		
beta testing	246		
types of	246		
acceptance test plan	283	291	
active design reviews	165		
activity diagrams	449		
actors	448	454	468
adjusted function point	313		
adjustment factor	313		
AFP	313		
alpha testing	75		
entry criteria	246		
exit criteria	246		
anomalies	246		
APFD	365		
cost-cognizant	367		
architectural design	238		
assessment	273		
authentication	238		
authorization	435		
automated script development	429		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		

automated testing	436		
behavioral view	447		
environmental view	448		
guidelines	429		
implementation view	448		
structural view	447		
user view	447		
automated testing tools	429	430	

B

backlog management index	389		
back tracking	508		
based applications	475		
baseline version	256		
basis path testing	138		
beta testing	247		
entry criteria	248		
exit criteria	247		
guidelines for	247		
beta-versions	90		
black-box testing	40	90	226
BMI	389		
bottom-up integration	224	225	
boundary value analysis	201		
boundary value checking	35		
brainstorm	222		
breadth first integration	37	222	
breakpoint	507		
conditional	507		
internal	507		
temporary	507		
unconditional	507		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		

bug classification 90
based on criticality 91
based on SDLC 421

C

call graph 227 228 229
capability maturity model 392
capacity testing 494
capture/playback tools 433
card-sorting technique 486
cause-effect graph 126
checklist 205
abstraction driven reading 205
function-point based scenarios 204
perspective-based reading 204
scenario-based reading 205
task driven reading 204
usage-based reading 192
checklists 67
class diagrams 448
classes 446 450
class-level testing 452
class-responsibility-collaboration 449
client-server 476 477
client-server model 475
cluster-level testing 452 464
CMM structure 392
code comprehension 432
code coverage 326
code traceability 292
coding 195
cohesion 136

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links	
collaboration diagrams	454	
commercial testing tools	437	
configuration testing	243	490
complexity analysis tools	432	
component diagrams	449	
concatenated loops	141	
confidentiality	238	
configuration/compatibility testing	490	
configuration management	435	
connection matrix	326	
content testing	487	
control flow	142	195
conventional testing	450	
coupling	355	
coverage	352	353
coverage analysis tools	433	
coverage criteria	262	
coverage identification problem	343	
coverage measures	284	
CPM	141	
CRC	449	
CRC model index card	449	
cross-site scripting	492	
customer problem metrics	388	
customer satisfaction metrics	388	

D

database servers	476
data complexity	331
data design	74

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
data flow	164	165	326
testing	164		
anomalies	165		
dataflow techniques	263		
data flow testing	166	352	
dynamic	169		
static	167		
terminology used in	166		
data-member	446		
DD graph	139		
dead code	288		
debuggers	510		
debugging	503	505	509
debugging techniques	506		
memory dump	506		
watch points	506		
decision node	139		
decision table	120		
extended entry	120		
limited entry	120		
test case design using	120		
decision table-based testing	119		
decision-to-decision-graph	139		
decomposition-based integration	219	228	
top-down	222		
types of	221		
decomposition tree	220		
defect age	323	341	
defect-arrival pattern during testing	389		
defect density	343	389	
defect density metrics	388		
defect fix time to retest	342		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		

defect rate 374
defect-removal efficiency 323
defect removal percentage 321
defect spoilage 324
defect-tracking 438
defect trend analysis 342
definition-clear path 189
definition node 222
definition-use path 256
delta build 256
delta version 222
deployment diagram 449
depth first 214
depth first integration 292
design traceability 255
development effort estimation 328
development ratio method 328
development testing 217
DFDs 508
distributed computing 476
DMAIC 399
DRE 389
driver module 228
drivers 214 322 362
dynamic binding 451
dynamic contents 488
dynamic slice 336
dynamic testing 431

E

earned value tracking 344
effectiveness of test cases 323

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links

EIF 310
encapsulation 446
equivalence classes 108
equivalence class testing 107
equivalent classes 108
    identification of 108
    valid 109
error 196
error detection efficiency 129
error guessing 195
error-prone modules 196
error-types 195
estimation of test cases 327
execution slice 361
exit criteria 321
external interface files 310

F

failure 33
failure impact 359
fan-in 332
fan-out 332
fault/defect/bug 33
fault-exposing-potential 358
fault-revealing test cases 263
FEP 358
finite state machine 115
fix backlog 389
fix response time 390
fix responsiveness 390
flow graph 139 197

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		

formal inspection 197
active design reviews 197
FTArm 197
Gilb inspection 197 200
Humphrey's inspection 197 201
N-fold inspections 197
phased inspection 197
structured walkthrough 205 333
FP 205
FPA 114
FSM 115
FTArm 219
functional decomposition 67 289
functional design 89
functional testing 232 289
function coverage 232 333
function point analysis 333
function points 81 309 335

G

Gantt charts 284
glass-box testing 135
goal question metric 318
GQM 318
graph matrix 156 157

H

Halstead product metrics 308
high-level design 67
hitting set 354
HTTP 239

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
human testing	188		

I

IBM rational SQA robot	438		
IFPUG	194	309	
incremental testing	221	461	
information flow complexity	332		
inheritance	447	451	460
inheritance testing	459		
inherited methods	460		
inspection process	191		
benefits of	194		
bug prevention	194		
bug reduction	194		
checklists	205	255	
cost of	196		
effectiveness of	196		
variants of	197		
inspections	190	381	
installation testing	82		
integration strategies	452		
integration testing	57	156	391
integration test plan	287		
integrity	75		
inter-class testing	464		
inter-cluster testing	464		
interface design	67		
interface testing	485	489	
internal design	310		
internal logical files	390		
internal testing	430		
intrusive tools	436		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links			
J				
junction node	139			
junior test engineers	276			
K				
KLOC	308			
KPAs	392	394	395	
L				
leaf nodes	219			
lines of code	308			
loadRunner	438			
load testing	240	241	438	494
load tests	327			
LOC	306			
private	307			
public metrics	307			
logic coverage	137			
condition coverage	136			
criteria	137			
decision/condition coverage	137			
decision or branch coverage	138			
multiple condition coverage	136			
statement coverage	161			
loop testing	67			
M				
maintenance testing	156			
master schedule	284			
master test plan	283			

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links	

maturity levels	393	413
mean-time to failure	388	
measurement program	318	
measurements	305	
memory testing tools	433	
MEP	229	230
method-level testing	452	
method-message path	464	
metrics	304	319
middleware	476	
Miller's theorem	142	
minimization techniques	263	
MM-path	230	231
module	76	
cohesion	76	
coupling	174	
MTTF	388	
multi-tiered applications	474	
mutants	174	
primary	177	
process	175	
secondary	174	

N

navigation testing	489	490
neighborhood integration	228	
nested loops	162	
network-testing tools	434	
N-fold inspection	202	
non-repudiation	238	
N-tier	476	

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links			

O

object	446	450		
object diagrams	449			
object model	445			
object-oriented modeling	447			
object-oriented technology	445			
object-oriented testing	450			

P

pair-wise integration	228			
path-based integration	229			
message	230			
module execution path	229			
sink node	229			
source node	229			
path testing	140			
independent path	140			
terminology	140			
unit testing	156			
use of	159			
percent delinquent fixes	390			
performance testing	238	239	240	489
	493			
PERT	284			
phased inspection	202			
polymorphism	447	451	452	
PORT	365			
prioritization	354	356		
based on requirements	364			
operational profiles	360			
regression test suite	356			

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links	
prioritization (*Cont.*)		
system test suite	356	
techniques	356	
prioritization scheme	355	
prioritized test suite	365	
probability of errors	331	
problems per user month	388	
procedure	335	
process improvement	195	
process maturity	396	
process quality	374	375
product quality	374	
program length	308	
program monitors	431	
program vocabulary	308	
program volume	309	
progressive testing	255	
project management	379	
project-staff ratio method	329	
PUM	388	

Q

quality assurance	377	
quality control	377	421
quality cost	375	
failure costs	376	
prevention costs	379	
quality factors	378	
quality management	381	

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links			

R

rate of fault detection	355			
reading techniques	203			
ad hoc method	203			
checklists	203			
realistic-size databases	240			
recovery testing	235	236		
recurrence ratio	342			
redundant test cases	353			
regions	139			
regression bugs	259			
regression number	258			
regression testability	257	258		
regression testing	256	257	258	505
	509	510		
objectives of	258			
problem	260			
techniques	260			
types	259			
regression test prioritization	265			
regression test problem	259			
regression test selection	263			
regression test suite	257			
regressive testing	255			
relevant slice	364			
relevant slices	360			
reliability	380			
requirement coverage matrix	290			
requirements	67			
gathering	91			
specification	120			

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
requirement traceability	292	326	
reviews	359	360	
risk analysis	325		

S

safe techniques	263		
sandwich integration testing	225		
scalability testing	494		
SDLC phases	66	212	429
security requirements	236		
security testing	237	238	491
security vulnerabilities	237		
selective retest technique	261		
sequence diagram	449		
severity of impact	359		
simple loops	162		
simulated testing	430		
sink nodes	229		
six sigma	399		
size metrics	308		
slicing technique	361		
dynamic slice	362		
execution slice	361		
relevant slice	364		
smoke test	290	324	
SMP	376		
software crisis	480		
software maintenance	261		
software measurement	305	307	

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
software metrics	304	306	
classification	306		
computed metrics	306		
definition of	306		
line of code	306		
objective	306		
primitive	306		
subjective metrics	306		
software quality	374	378	384
software quality metrics	387	388	
software testing	5		
as a process	22		
economics of	39		
effective	16		
evolution of	5		
exhaustive	16		
goals of	10		
methodology	51		
model for	15		
myths and facts	8		
psychology for	13		
schools of	23		
STLC	46		
strategy	52		
techniques	57		
terminology	33		
testing definitions	14		
software tools	430		
spanning set	326		
specialized environments	443		
spoilage	324		
SQA	378		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
SQA activities	381		
SQA audit	382		
SQA group	421		
SQA models	390		
SQA team	381		
SQL injection	492		
SRS	239		
staff productivity	321		
state-based testing	457	458	
state chart diagrams	449		
state graph	115		
state table	116		
state table-based testing	114	116	
static contents	488		
static program analysers	431		
static testing	189	381	
benefits	58		
objectives	189		
types of	189		
static testing tools	431		
static test points	135	338	
STQM	397		
stress testing	241	494	
structural complexity	57		
structured walkthrough	322		
stubs	215	220	288
subclass	460		
superclass	459	460	
suspension criteria for testing	255		
SUT	321		
SVVP	213		
symptoms	505	508	

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		

system complexity 284
system-level testing 452
system testing 233 290 331
system test plan 289

T

task planning method 330
TCE 345
technical review 206
template class 451
test 34
  case 34
  design 47
  execution 49
  factors 52
  incident 35
  phase 52
  planning 47
  strategy matrix 52
  test oracle 35
  testware 35
testability 380
test automation 429
test case effectiveness metric 345
test case generator 433
test case prioritization 355
  coverage-based 356
  general test case prioritization 355
  types of 355
  version-specific 355
test case specifications 273
test coverage 326

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
test data generator	433		
test design	215		
TestDirector	438		
TestStudio	434		
test driver	274		
test effort estimation	335		
test enablers	430		
test engineers	274	276	
tester productivity measures	344		
testers	275	276	
test execution and evaluation tools	433		
test harness	286	288	296
test incident report	43		
testing	55		
life cycle model	59		
measurement objectives	318		
methodologies	43		
principles	57		
tactics	58		
tools	57		
unit testing	274		
testing activity tools	432		
testing cost estimation	322		
testing efforts	327	332	
testing group	275		
testing improvement model	406		
testing metrics	341		
testing of OO classes	455		
feature-based testing of classes	455		
role of invariants in class testing	456		
state-based testing	457		
testing feature groups	456		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		
testing of web-based systems	474	484	
testing techniques	295	417	
testing tools	429	430	431
categorization	431		
costs incurred	435		
selection	434		
test leader	273		
test log	273		
test management	275		
key elements of	273		
test management tools	434		
test manager	273		
test maturity matrix	409	412	413
test maturity models	404	406	413
test monitoring	274		
test organization	80	418	
test organization model	408		
test plan	80		
acceptance	80		
function	80		
integration	80		
system	277		
unit	277		
test plan	212	398	
components	282		
hierarchy	228		
test planning	295	417	
test point analysis	335		
test procedure method	295	329	330
test process	406		
test process management	412		
test process maturity	405		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links			
test process maturity models	405	406		
test process optimization	421			
test result specifications	274			
test script language	437			
test sessions	292			
test specialist	273			
test specifications	222	262		
test suite	296	298	352	354
test suite prioritization	354			
test suites	353			
test summary report	308			
test team size	330			
test tools	338			
testware	339			
testware management	412			
TIM	406			
TMM	405			
assessment model	422			
components	413			
TMM levels	415			
token count	73			
TOM level	408			
top-down integration	222			
total quality management	73			
total test sessions	335			
TPA	397			
TPI	405			
TPI model	409			
TQM	232			
traceability	289			
backward	320			
forward	437			

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links			

U

UFP	312			
UML	447	466	467	468
UML-based modeling	481			
conceptual modeling	481			
configuration modeling	483			
navigation modeling	481			
presentation modeling	482			
task modeling	482			
web scenarios modeling	482			
UML-based OO testing	466			
UML diagrams	467			
UML modeling	448			
unadjusted function point	312			
unified modeling language	447			
unit testing	213	331		
unit test plan	286			
unit validation testing	78			
unit verification	163			
unstructured loops	242			
usability testing	166	486	487	
usage node	166			
computation	166			
predicate	449			
use-case diagram	448	454	467	
use-case model	448			
actors	448			
use-case diagrams	448			
use-case templates	454			

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links		

V

VAF	313		
validation	54	284	419
activities	79		
test plan	79		
validation activities	282	454	
validation plan	284		
validation test execution	212		
validation testing	66	284	
validation test plan	285		
value	313		
verification	70	77	282
activities	75		
of architectural design	75		
of data design	74		
of high-level design	76		
of low-level design	71		
of objectives	70		
of requirements	76		
of user-interface design	74		
verification test plan	285		
verify	77		
code	74		
high-level design	77		
low-level design	71		
V-testing model	56	69	
V&V diagram	285	378	

W

walkthroughs	296	381	419
watch points	506		

This page has been reformatted by Knovel to provide easier navigation.

Index Terms	Links			
WBS	280			
web applications	438	452	474	475
web-based applications	475			
web-based software	476			
web browsers	477			
web crisis	480			
web-enabled applications	475			
web engineering	480			
web page	475			
website	475			
web software	478			
web technology	475			
3-tier	475			
first generation/2-tier	475			
N-tier	475			
white-box testing	58	417		
WinRunner	437			
work breakdown structure	280	322		
worst-case testing method	92			

X

XSS	492

This page has been reformatted by Knovel to provide easier navigation.

Paul AMMANN & Jeff OFFUTT

INTRODUCTION TO SOFTWARE TESTING

CAMBRIDGE www.cambridge.org/9780521880381

This page intentionally left blank

Introduction to Software Testing

Extensively class tested, this text takes an innovative approach to software testing: it defines testing as the process of applying a few well-defined, general-purpose test criteria to a structure or model of the software. The structure of the text directly reflects the pedagogical approach and incorporates the latest innovations in testing, including modern types of software such as OO, Web applications, and embedded software. The book contains numerous examples throughout. An instructor's solution manual, PowerPoint slides, sample syllabi, additional examples and updates, testing tools for students, and example software programs in Java are available on an extensive Web site at www.introsoftwaretesting.com.

Paul Ammann, PhD, is an Associate Professor of software engineering at George Mason University. He received an outstanding teaching award in 2007 from the Volgenau School of Information Technology and Engineering. Dr. Ammann earned an AB degree in computer science from Dartmouth College and MS and PhD degrees in computer science from the University of Virginia.

Jeff Offutt, PhD, is a Professor of software engineering at George Mason University. He is editor-in-chief of the *Journal of Software Testing, Verification and Reliability*; chair of the steering committee for the IEEE International Conference on Software Testing, Verification, and Validation; and on the editorial boards for several journals. He recived the outstanding teacher award from the Volgenau School of Information Technology and Engineering in 2003. Dr. Offutt earned a BS degree in mathematics and data processing from Morehead State University and MS and PhD degrees in computer science from the Georgia Institute of Technology.

INTRODUCTION TO SOFTWARE TESTING

Paul Ammann
George Mason University

Jeff Offutt
George Mason University

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK
Published in the United States of America by Cambridge University Press, New York
www.cambridge.org
Information on this title: www.cambridge.org/9780521880381

© Paul Ammann and Jeff Offutt 2008

This publication is in copyright. Subject to statutory exception and to the provision of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published in print format 2008

ISBN-13 978-0-511-39330-3 eBook (EBL)

ISBN-13 978-0-521-88038-1 hardback

Cambridge University Press has no responsibility for the persistence or accuracy of urls for external or third-party internet websites referred to in this publication, and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

Contents

List of Figures *page* ix
List of Tables xiii
Preface xv

Part 1 Overview 1

1 Introduction 3
1.1 Activities of a Test Engineer 4
1.1.1 Testing Levels Based on Software Activity 5
1.1.2 Beizer's Testing Levels Based on Test Process Maturity 8
1.1.3 Automation of Test Activities 10
1.2 Software Testing Limitations and Terminology 11
1.3 Coverage Criteria for Testing 16
1.3.1 Infeasibility and Subsumption 20
1.3.2 Characteristics of a Good Coverage Criterion 20
1.4 Older Software Testing Terminology 21
1.5 Bibliographic Notes 22

Part 2 Coverage Criteria 25

2 Graph Coverage 27
2.1 Overview 27
2.2 Graph Coverage Criteria 32
2.2.1 Structural Coverage Criteria 33
2.2.2 Data Flow Criteria 44
2.2.3 Subsumption Relationships among Graph Coverage Criteria 50
2.3 Graph Coverage for Source Code 52

2.3.1 Structural Graph Coverage for Source Code 52
2.3.2 Data Flow Graph Coverage for Source Code 54
2.4 Graph Coverage for Design Elements 65
2.4.1 Structural Graph Coverage for Design Elements 65
2.4.2 Data Flow Graph Coverage for Design Elements 67
2.5 Graph Coverage for Specifications 75
2.5.1 Testing Sequencing Constraints 75
2.5.2 Testing State Behavior of Software 77
2.6 Graph Coverage for Use Cases 87
2.6.1 Use Case Scenarios 90
2.7 Representing Graphs Algebraically 91
2.7.1 Reducing Graphs to Path Expressions 94
2.7.2 Applications of Path Expressions 96
2.7.3 Deriving Test Inputs 96
2.7.4 Counting Paths in a Flow Graph and Determining Max Path Length 97
2.7.5 Minimum Number of Paths to Reach All Edges 98
2.7.6 Complementary Operations Analysis 98
2.8 Bibliographic Notes 100

3 Logic Coverage 104

3.1 Overview: Logic Predicates and Clauses 104
3.2 Logic Expression Coverage Criteria 106
3.2.1 Active Clause Coverage 107
3.2.2 Inactive Clause Coverage 111
3.2.3 Infeasibility and Subsumption 112
3.2.4 Making a Clause Determine a Predicate 113
3.2.5 Finding Satisfying Values 115
3.3 Structural Logic Coverage of Programs 120
3.3.1 Predicate Transformation Issues 127
3.4 Specification-Based Logic Coverage 131
3.5 Logic Coverage of Finite State Machines 134
3.6 Disjunctive Normal Form Criteria 138
3.7 Bibliographic Notes 147

4 Input Space Partitioning 150

4.1 Input Domain Modeling 152
4.1.1 Interface-Based Input Domain Modeling 153
4.1.2 Functionality-Based Input Domain Modeling 154
4.1.3 Identifying Characteristics 154
4.1.4 Choosing Blocks and Values 156
4.1.5 Using More than One Input Domain Model 158
4.1.6 Checking the Input Domain Model 158
4.2 Combination Strategies Criteria 160
4.3 Constraints among Partitions 165
4.4 Bibliographic Notes 166

5 Syntax-Based Testing 170
5.1 Syntax-Based Coverage Criteria 170
5.1.1 BNF Coverage Criteria 170
5.1.2 Mutation Testing 173
5.2 Program-Based Grammars 176
5.2.1 BNF Grammars for Languages 176
5.2.2 Program-Based Mutation 176
5.3 Integration and Object-Oriented Testing 191
5.3.1 BNF Integration Testing 192
5.3.2 Integration Mutation 192
5.4 Specification-Based Grammars 197
5.4.1 BNF Grammars 198
5.4.2 Specification-Based Mutation 198
5.5 Input Space Grammars 201
5.5.1 BNF Grammars 201
5.5.2 Mutation for Input Grammars 204
5.6 Bibliographic Notes 210

Part 3 Applying Criteria in Practice 213

6 Practical Considerations 215
6.1 Regression Testing 215
6.2 Integration and Testing 217
6.2.1 Stubs and Drivers 218
6.2.2 Class Integration Test Order 218
6.3 Test Process 219
6.3.1 Requirements Analysis and Specification 220
6.3.2 System and Software Design 221
6.3.3 Intermediate Design 222
6.3.4 Detailed Design 223
6.3.5 Implementation 223
6.3.6 Integration 224
6.3.7 System Deployment 224
6.3.8 Operation and Maintenance 224
6.3.9 Summary 225
6.4 Test Plans 225
6.5 Identifying Correct Outputs 230
6.5.1 Direct Verification of Outputs 230
6.5.2 Redundant Computations 231
6.5.3 Consistency Checks 231
6.5.4 Data Redundancy 232
6.6 Bibliographic Notes 233

7 Engineering Criteria for Technologies 235
7.1 Testing Object-Oriented Software 236
7.1.1 Unique Issues with Testing OO Software 237

7.1.2 Types of Object-Oriented Faults 237
7.2 Testing Web Applications and Web Services 256
7.2.1 Testing Static Hyper Text Web Sites 257
7.2.2 Testing Dynamic Web Applications 257
7.2.3 Testing Web Services 260
7.3 Testing Graphical User Interfaces 260
7.3.1 Testing GUIs 261
7.4 Real-Time Software and Embedded Software 262
7.5 Bibliographic Notes 265

8 Building Testing Tools 268

8.1 Instrumentation for Graph and Logical Expression Criteria 268
8.1.1 Node and Edge Coverage 268
8.1.2 Data Flow Coverage 271
8.1.3 Logic Coverage 272
8.2 Building Mutation Testing Tools 272
8.2.1 The Interpretation Approach 274
8.2.2 The Separate Compilation Approach 274
8.2.3 The Schema-Based Approach 275
8.2.4 Using Java Reflection 276
8.2.5 Implementing a Modern Mutation System 277
8.3 Bibliographic Notes 277

9 Challenges in Testing Software 280

9.1 Testing for Emergent Properties: Safety and Security 280
9.1.1 Classes of Test Cases for Emergent Properties 283
9.2 Software Testability 284
9.2.1 Testability for Common Technologies 285
9.3 Test Criteria and the Future of Software Testing 286
9.3.1 Going Forward with Testing Research 288
9.4 Bibliographic Notes 290

List of Criteria 293
Bibliography 295
Index 319

List of Figures

1.1 Activities of test engineers *page* 4
1.2 Software development activities and testing levels – the "V Model" 6
2.1 Graph (a) has a single initial node, graph (b) multiple initial nodes, and graph (c) (rejected) with no initial nodes 28
2.2 Example of paths 29
2.3 A single entry single exit graph 30
2.4 Test case mappings to test paths 31
2.5 A set of test cases and corresponding test paths 32
2.6 A graph showing node coverage and edge coverage 34
2.7 Two graphs showing prime path coverage 37
2.8 Graph with a loop 37
2.9 Tours, sidetrips, and detours in graph coverage 38
2.10 An example for prime test paths 40
2.11 A graph showing variables, def sets and use sets 44
2.12 A graph showing an example of du-paths 46
2.13 Graph showing explicit def and use sets 47
2.14 Example of the differences among the three data flow coverage criteria 49
2.15 Subsumption relations among graph coverage criteria 50
2.16 CFG fragment for the if-else structure 52
2.17 CFG fragment for the if structure without an else 53
2.18 CFG fragment for the while loop structure 53
2.19 CFG fragment for the for loop structure 54
2.20 CFG fragment for the case structure 54
2.21 TestPat for data flow example 56
2.22 A simple call graph 65
2.23 A simple inheritance hierarchy 66
2.24 An inheritance hierarchy with objects instantiated 67
2.25 An example of parameter coupling 68
2.26 Coupling du-pairs 69
2.27 Last-defs and first-uses 69

2.28 Quadratic root program 71
2.29 Def-use pairs under intra-procedural and inter-procedural data flow 72
2.30 Def-use pairs in object-oriented software 72
2.31 Def-use pairs in web applications and other distributed software 73
2.32 Control flow graph using the File ADT 76
2.33 Elevator door open transition 79
2.34 Stutter – Part A 80
2.35 Stutter – Part B 81
2.36 A FSM representing Stutter, based on control flow graphs of the methods 82
2.37 A FSM representing Stutter, based on the structure of the software 83
2.38 A FSM representing Stutter, based on modeling state variables 84
2.39 A FSM representing Stutter, based on the specifications 85
2.40 Class Queue for exercises. 86
2.41 ATM actor and use cases 88
2.42 Activity graph for ATM withdraw funds 90
2.43 Examples of path products 92
2.44 Null path that leads to additive identity ϕ 93
2.45 A or lambda 94
2.46 Example graph to show reduction to path expressions 94
2.47 After step 1 in path expression reduction 95
2.48 After step 2 in path expression reduction 95
2.49 After step 3 in path expression reduction 95
2.50 Removing arbitrary nodes 95
2.51 Eliminating node n_2 95
2.52 Removing sequential edges 95
2.53 Removing self-loop edges 96
2.54 Final graph with one path expression 96
2.55 Graph example for computing maximum number of paths 97
2.56 Graph example for complementary path analysis 99
3.1 Subsumption relations among logic coverage criteria 113
3.2 TriTyp – Part A 121
3.3 TriTyp – Part B 122
3.4 Calendar method 132
3.5 FSM for a memory car seat – Lexus 2003 ES300 135
3.6 Fault detection relationships 143
4.1 Partitioning of input domain D into three blocks 151
4.2 Subsumption relations among input space partitioning criteria 163
5.1 Method Min and six mutants 177
5.2 Mutation testing process 181
5.3 Partial truth table for $(a \wedge b)$ 187
5.4 Finite state machine for SMV specification 199
5.5 Mutated finite state machine for SMV specification 200
5.6 Finite state machine for bank example 202
5.7 Finite state machine for bank example grammar 202
5.8 Simple XML message for books 204
5.9 XML schema for books 205

7.1 Example class hierarchy in UML 238
7.2 Data flow anomalies with polymorphism 238
7.3 Calls to d() when object has various actual types 239
7.4 ITU: Descendant with no overriding methods 241
7.5 SDA, SDIH: State definition anomalies 243
7.6 IISD: Example of indirect inconsistent state definition 244
7.7 ACB1: Example of anomalous construction behavior 245
7.8 SVA: State visibility anomaly 247
7.9 Sample class hierarchy (a) and associated type families (b) 248
7.10 Control flow graph fragment (a) and associated definitions and uses (b) 249
7.11 Def-use pairs in object-oriented software 250
7.12 Control flow schematic for prototypical coupling sequence 251
7.13 Sample class hierarchy and *def-use* table 252
7.14 Coupling sequence: *o* of type *A* (a) bound to instance of *A* (b), *B* (c) or *C* (d) 253
8.2 Node coverage instrumentation 269
8.3 Edge coverage instrumentation 270
8.4 All uses coverage instrumentation 271
8.5 Correlated active clause coverage instrumentation 273

List of Tables

2.1 Defs and uses at each node in the CFG for TestPat *page* 57
2.2 Defs and uses at each edge in the CFG for TestPat 57
2.3 Du-path sets for each variable in TestPat 58
2.4 Test paths to satisfy all du-paths coverage on TestPat 59
2.5 Test paths and du-paths covered on TestPat 59
3.1 Reachability for Triang predicates 123
3.2 Reachability for Triang predicates – reduced by solving for triOut 124
3.3 Predicate coverage for Triang 125
3.4 Clause coverage for Triang 126
3.5 Correlated active clause coverage for Triang 127
3.6 Correlated active clause coverage for cal() preconditions 133
3.7 Predicates from memory seat example 136
3.8 DNF fault classes 143
4.1 First partitioning of TriTyp's inputs (interface-based) 156
4.2 Second partitioning of TriTyp's inputs (interface-based) 157
4.3 Possible values for blocks in the second partitioning in Table 4.2 157
4.4 Geometric partitioning of TriTyp's inputs (functionality-based) 158
4.5 Correct geometric partitioning of TriTyp's inputs (functionality-based) 158
4.6 Possible values for blocks in geometric partitioning in Table 4.5 159
4.7 Examples of invalid block combinations 165
5.1 Java's access levels 193
6.1 Testing objectives and activities during requirements analysis and specification 221
6.2 Testing objectives and activities during system and software design 222
6.3 Testing objectives and activities during intermediate design 222
6.4 Testing objectives and activities during detailed design 223
6.5 Testing objectives and activities during implementation 223
6.6 Testing objectives and activities during integration 224
6.7 Testing objectives and activities during system deployment 224
6.8 Testing objectives and activities during operation and maintenance 225
7.1 Faults and anomalies due to inheritance and polymorphism 240

7.2 ITU: Code example showing inconsistent type usage 242
7.3 IC: Incomplete construction of state variable ***fd*** 246
7.4 Summary of sample coupling paths 254
7.5 Binding triples for coupling sequence from class hierarchy in Figure 7.13 254

Preface

This book presents software testing as a practical engineering activity, essential to producing high-quality software. It is designed to be used as the primary textbook in either an undergraduate or graduate course on software testing, as a supplement to a general course on software engineering or data structures, and as a resource for software test engineers and developers. This book has a number of unique features:

- It organizes the complex and confusing landscape of test coverage criteria with a novel and extremely simple structure. At a technical level, software testing is based on satisfying coverage criteria. The book's central observation is that there are few truly different coverage criteria, each of which fits easily into one of four categories: graphs, logical expressions, input space, and syntax structures. This not only simplifies testing, but it also allows a convenient and direct theoretical treatment of each category. This approach contrasts strongly with the traditional view of testing, which treats testing at each phase in the development process differently.
- It is designed and written to be a textbook. The writing style is direct, it builds the concepts from the ground up with a minimum of required background, and it includes lots of examples, homework problems, and teaching materials. It provides a balance of theory and practical application, presenting testing as a collection of objective, quantitative activities that can be measured and repeated. The theoretical concepts are presented when needed to support the practical activities that test engineers follow.
- It assumes that testing is part of a mental discipline that helps all IT professionals develop higher-quality software. Testing is not an anti-engineering activity, and it is not an inherently destructive process. Neither is it only for testing specialists or domain experts who know little about programming or math.
- It is designed with modular, interconnecting pieces; thus it can be used in multiple courses. Most of the book requires only basic discrete math and introductory programming, and the parts that need more background are clearly marked. By

using the appropriate sections, this book can support several classes, as described later in the preface.

- It assumes the reader is learning to be an engineer whose goal is to produce the best possible software with the lowest possible cost. The concepts in this book are well grounded in theory, are practical, and most are currently in use.

WHY SHOULD THIS BOOK BE USED?

Not very long ago, software development companies could afford to employ programmers who could not test and testers who could not program. For most of the industry, it was not necessary for either group to know the technical principles behind software testing or even software development. Software testing in industry historically has been a nontechnical activity. Industry viewed testing primarily from the managerial and process perspective and had limited expectations of practitioners' technical training.

As the software engineering profession matures, and as software becomes more pervasive in everyday life, there are increasingly stringent requirements for software reliability, maintainability, and security. Industry must respond to these changes by, among other things, improving the way software is tested. This requires increased technical expertise on the part of test engineers, as well as increased emphasis on testing by software developers. The good news is that the knowledge and technology are available and based on over 30 years of research and practice. This book puts that knowledge into a form that students, test engineers, test managers, and developers can access.

At the same time, it is relatively rare to find courses that teach testing in universities. Only a few undergraduate courses exist, almost no masters degree programs in computer science or software engineering require a course in software testing, and only a few dozen have an elective course. Not only is testing not covered as an essential part of undergraduate computer science education, most computer science students either never gain any knowledge about testing, or see only a few lectures as part of a general course in software engineering.

The authors of this book have been teaching software testing to software engineering and computer science students for more than 15 years. Over that time we somewhat reluctantly came to the conclusion that no one was going to write the book we wanted to use. Rather, to get the book we wanted, we would have to write it.

Previous testing books have presented software testing as a relatively simple subject that relies more on process than technical understanding of how software is constructed, as a complicated and fractured subject that requires detailed understanding of numerous software development technologies, or as a completely theoretical subject that can be mastered only by mathematicians and theoretical computer scientists. Most books on software testing are organized around the phases in a typical software development lifecycle, an approach that has the unfortunate side effect of obscuring common testing themes. Finally, most testing books are written as reference books, not textbooks. As a result, only instructors with prior expertise in software testing can easily teach the subject. **This book is accessible to instructors who are not already testing experts**.

This book differs from other books on software testing in other important ways. Many books address managing the testing process. While this is important, it is equally important to give testers specific techniques grounded in basic theory. This book provides a balance of theory and practical application. This is important information that software companies must have; however, this book focuses specifically on the technical nuts-and-bolts issues of designing and creating tests. Other testing books currently on the market focus on specific techniques or activities, such as system testing or unit testing. This book is intended to be comprehensive over the entire software development process and to cover as many techniques as possible.

As stated previously, the motivation for this book is to support courses in software testing. Our first target was our own software testing course in our Software Engineering MS program at George Mason University. This popular elective is taught to about 30 computer science and software engineering students every semester. We also teach PhD seminars in software testing, industry short courses on specialized aspects, and lectures on software testing in various undergraduate courses. Although few undergraduate courses on software testing exist, we believe that they should exist, and we expect they will in the near future. Most testing books are not designed for classroom use. We specifically wrote this book to support our classroom activities, and it is no accident that the syllabus for our testing course, available on the book's Web site (www.introsoftwaretesting.com), closely follows the table of contents for this book.

This book includes numerous carefully worked examples to help students and teachers alike learn the sometimes complicated concepts. The instructor's resources include high-quality powerpoint slides, presentation hints, solutions to exercises, and working software. Our philosophy is that we are doing more than writing a book; we are offering our course to the community. One of our goals was to write material that is scholarly and true to the published research literature, but that is also accessible to nonresearchers. Although the presentation in the book is quite a bit different from the research papers that the material is derived from, the essential ideas are true to the literature. To make the text flow more smoothly, we have removed the references from the presentation. For those interested in the research genealogy, each chapter closes with a bibliographic notes section that summarizes where the concepts come from.

WHO SHOULD READ THIS BOOK?

Students who read and use this book will learn the fundamental principles behind software testing, and how to apply these principles to produce better software, faster. They will not only become better programmers, they will also be prepared to carry out high-quality testing activities for their future employers. *Instructors* will be able to use this book in the classroom, even without prior practical expertise in software testing. The numerous exercises and thought-provoking problems, classroom-ready and classroom-tested slides, and suggested outside activities make this material teachable by instructors who are not already experts in software testing. *Research students* such as beginning PhD students will find this book to be an invaluable resource as a starting point to the field. The theory is sound and clearly

presented, the practical applications reveal what is useful and what is not, and the advanced reading and bibliographic notes provide pointers into the literature. Although the set of research students in software testing is a relatively small audience, we believe it is a key audience, because a common, easily achievable baseline would reduce the effort required for research students to join the community of testing researchers. *Researchers* who are already familiar with the field will find the criteria-approach to be novel and interesting. Some may disagree with the pedagogical approach, but we have found that the view that testing is an application of only a few criteria to a very few software structures to be very helpful to our research. We hope that testing research in the future will draw away from searches for more criteria to novel uses and evaluations of existing criteria.

Testers in the industry will find this book to be an invaluable collection of techniques that will help improve their testing, no matter what their current process is. The criteria presented here are intended to be used as a "toolbox" of tricks that can be used to find faults. *Developers* who read this book will find numerous ways to improve their own software. Their self-testing activities can become more efficient and effective, and the discussions of software faults that test engineers search for will help developers avoid them. To paraphrase a famous parable, if you want to teach a person to be a better fisherman, explain how and where the fish swim. Finally, *managers* will find this book to be a useful explanation of how clever test engineers do their job, and of how test tools work. They will be able to make more effective decisions regarding hiring, promotions, and purchasing tools.

HOW CAN THIS BOOK BE USED?

A major advantage of the structure of this book is that it can be easily used for several different courses. Most of the book depends on material that is taught very early in college and some high schools: basic concepts from data structures and discrete math. The sections are organized so that the early material in each chapter is accessible to less advanced students, and material that requires more advanced knowledge is clearly marked.

Specifically, the book defines six separate sets of chapter sections that form *streams* through the book:

1. A module within a CS II course
2. A sophomore-level course on software testing
3. A module in a general software engineering course
4. A senior-level course on software testing
5. A first-year MS level course on software testing
6. An advanced graduate research-oriented course on software testing
7. Industry practioner relevant sections

The stream approach is illustrated in the abbreviated table of contents in the figure shown on pp. xix–xx. Each chapter section is marked with which stream it belongs too. Of course, individual instructors, students, and readers may prefer to adapt the stream to their own interests or purposes. We suggest that the first two sections of Chapter 1 and the first two sections of Chapter 6 are appropriate reading for a module in a data structures (CS II) class, to be followed by a simple

Stream 1: Module in a CS II course.

Stream 2: Sophomore-level course on software testing.

Stream 3: Module in a general software engineering course.

Stream 4: Senior-level course on software testing.

Stream 5: First-year MS course on software testing.

Stream 6: Advanced graduate research-oriented course on software testing.

Stream 7: Industry practitioner relevant sections

	STREAMS						
	1	2	3	4	5	6	7
Part I: Overview							
Chapter 1. Introduction	■	▬	▬	▬	▭	▭	▭
1.1 Activities of a Test Engineer	■	▬	▬	▬	▭	▭	▭
1.2 Software Testing Limitations and Terminology	■	▬	▬	▬	▭	▭	▭
1.3 Coverage Criteria for Testing		▬	▬	▬	▭	▭	▭
1.4 Older Software Testing Terminology					▭	▭	
1.5 Bibliographic Notes						▭	
Part II: Coverage Criteria							
Chapter 2. Graph Coverage		▬	▬	▬	▭	▭	▭
2.1 Overview		▬	▬	▬	▭	▭	▭
2.2 Graph Coverage Criteria		▬	▬	▬	▭	▭	▭
2.3 Graph Coverage for Source Code		▬	▬	▬	▭	▭	▭
2.4 Graph Coverage for Design Elements				▬	▭	▭	▭
2.5 Graph Coverage for Specifications				▬	▭	▭	▭
2.6 Graph Coverage for Use Cases				▬	▭	▭	▭
2.7 Representing Graphs Algebraically					▭	▭	
2.8 Bibliographic Notes						▭	
Chapter 3. Logic Coverage		▬	▬	▬	▭	▭	▭
3.1 Overview: Logic Predicates and Clauses		▬	▬	▬	▭	▭	▭
3.2 Logic Expression Coverage Criteria		▬	▬	▬	▭	▭	▭
3.3 Structural Logic Coverage of Programs		▬		▬	▭	▭	
3.4 Specification-Based Logic Coverage				▬	▭	▭	
3.5 Logic Coverage of Finite State Machines			▬	▬	▭	▭	
3.6 Disjunctive Normal Form Criteria					▭	▭	
3.7 Bibliographic Notes						▭	
Chapter 4. Input Space Partitioning			▬	▬	▭	▭	▭
4.1 Input Domain Modeling			▬	▬	▭	▭	▭
4.2 Combination Strategies Criteria			▬	▬	▭	▭	▭
4.3 Constraints among Partitions					▭	▭	▭
4.4 Bibliographic Notes						▭	
Chapter 5. Syntax-Based Testing		▬	▬	▬	▭	▭	▭
5.1 Syntax-Based Coverage Criteria		▬	▬	▬	▭	▭	▭
5.2 Program-Based Grammars			▬	▬	▭	▭	▭
5.3 Integration and Object-Oriented Testing				▬	▭	▭	
5.4 Specification-Based Grammars					▭	▭	
5.5 Input Space Grammars		▬	▬	▬	▭	▭	▭
5.6 Bibliographic Notes						▭	

Stream 1: Module in a CS II course.

Stream 2: Sophomore-level course on software testing.

Stream 3: Module in a general software engineering course.

Stream 4: Senior-level course on software testing.

Stream 5: First-year MS course on software testing.

Stream 6: Advanced graduate research-oriented course on software testing.

Stream 7: Industry practitioner relevant sections

	STREAMS						
	1	2	3	4	5	6	7
Part III: Applying Criteria in Practice							
Chapter 6. Practical Considerations	■	▬	▬	▬	▭	▭	▭
6.1 Regression Testing	■	▬	▬	▬	▭	▭	▭
6.2 Integration and Testing	■	▬	▬	▬	▭	▭	▭
6.3 Test Process			▬	▬	▭	▭	▭
6.4 Test Plans			▬	▬	▭	▭	▭
6.5 Identifying Correct Outputs		▬	▬	▬	▭	▭	
6.5 Bibliographic Notes						▭	
Chapter 7. Engineering Criteria for Technologies				▬	▭	▭	▭
7.1 Testing Object-Oriented Software				▬	▭	▭	▭
7.2 Testing Web Applications and Web Services					▭	▭	▭
7.3 Testing Graphical User Interfaces					▭	▭	▭
7.4 Real-Time Software and Embedded Software					▭	▭	▭
7.5 Bibliographic Notes						▭	
Chapter 8. Building Testing Tools					▭	▭	▭
8.1 Instrumentation for Graph and Logical Expression Criteria					▭	▭	▭
8.2 Building Mutation Testing Tools						▭	
8.3 Bibliographic Notes						▭	
Chapter 9. Challenges in Testing Software					▭	▭	▭
9.1 Testing for Emergent Properties: Safety and Security					▭	▭	▭
9.2 Software Testability					▭	▭	▭
9.3 Test Criteria and the Future of Software Testing					▭	▭	
9.4 Bibliographic Notes						▭	

assignment. Our favorite is to ask the students to retrieve one of their previously graded programs and satisfy some simple test criterion like branch coverage. We offer points for every fault found, driving home two concepts: an "A" grade doesn't mean the program always works, and finding faults is a good thing.

The sophomore-level course on software testing (stream 2) is designed to immediately follow a data structures course (CS II). The marked sections contain material that depends only on data structures and discrete math.

A module in a general software engineering course (stream 3) could augment the survey material typical in such courses. The sections marked provide basic literacy in software testing.

The senior-level course on software testing (stream 4) is the primary target for this text. It adds material that requires a little more sophistication in terms of

software development than the sophomore stream. This includes sections in Chapter 2 on data flow testing, sections that involve integration testing of multiple modules, and sections that rely on grammars or finite state machines. Most senior computer science students will have seen this material in their other courses. Most of the sections that appear in stream 4 but not stream 2 could be added to stream 2 with appropriate short introductions. It is important to note that a test engineer does not need to know all the theory of parsing to use data flow testing or all the theory on finite state machines to use statecharts for testing.

The graduate-level course on software testing (stream 5) adds some additional sections that rely on a broader context and that require more theoretical maturity. For example, these sections use knowledge of elementary formal methods, polymorphism, and some of the UML diagrams. Some of the more advanced topics and the entire chapter on building testing tools are also intended for a graduate audience. This chapter could form the basis for a good project, for example, to implement a simple coverage analyzer.

An advanced graduate course in software testing with a research emphasis such as a PhD seminar (stream 6) includes issues that are still unproven and research in nature. The bibliographic notes are recommended only for these students as indicators for future in-depth reading.

Finally, sections that are reasonably widely used in industry, especially those that have commercial tool support, are marked for stream 7. These sections have a minimum of theory and omit criteria that are still of questionable usefulness.

Extensive supplementary materials, including sample syllabuses, PowerPoint slides, presentation hints, solutions to exercises, working software, and errata are available on the book's companion Web site.

ACKNOWLEDGMENTS

Many people helped us write this book. Not only have the students in our Software Testing classes at George Mason been remarkably tolerant of using a work in progress, they have enthusiastically provided feedback on how to improve the text. We cannot acknowledge all by name (ten semesters worth of students have used it!), but the following have made especially large contributions: Aynur Abdurazik, Muhammad Abdulla, Yuquin Ding, Jyothi Chinman, Blaine Donley, Patrick Emery, Brian Geary, Mark Hinkle, Justin Hollingsworth, John King, Yuelan Li, Xiaojuan Liu, Chris Magrin, Jyothi Reddy, Raimi Rufai, Jeremy Schneider, Bill Shelton, Frank Shukis, Quansheng Xiao, and Linzhen Xue. We especially appreciate those who generously provided extensive comments on the entire book: Guillermo Calderon-Meza, Becky Hartley, Gary Kaminski, and Andrew J. Offutt. We gratefully acknowledge the feedback of early adopters at other educational institutions: Roger Alexander, Jane Hayes, Ling Liu, Darko Marinov, Arthur Reyes, Michael Shin, and Tao Xie. We also want to acknowledge several people who provided material for the book: Roger Alexander, Mats Grindal, Hong Huang, Gary Kaminski, Robert Nilsson, Greg Williams, Wuzhi Xu. We were lucky to receive excellent suggestion from Lionel Briand, Renée Bryce, Kim King, Sharon Ritchey, Bo Sanden, and Steve Schach. We are grateful to our editor, Heather Bergman,

for providing unwavering support and enforcing the occasional deadline to move the project along, as well as Kerry Cahill from Cambridge University Press for very strong support on this project.

We also acknowledge George Mason University for supporting both of us on sabbaticals and for providing GTA support at crucial times. Our department Chair, Hassan Gomaa, has enthusiastically supported this effort.

Finally, of course none of this is possible without the support of our families. Thanks to Becky, Jian, Steffi, Matt, Joyce, and Andrew for keeping us grounded in reality and helping keep us happy for the past five years.

Just as all programs contain faults, all texts contain errors. Our text is no different. And, as responsibility for software faults rests with the developers, responsibility for errors in this text rests with us, the authors. In particular, the bibliographic notes sections reflect our perspective of the testing field, a body of work we readily acknowledge as large and complex. We apologize in advance for omissions, and invite pointers to relevant citations.

Paul Ammann
Jeff Offutt

PART 1

Overview

1

Introduction

The ideas and techniques of software testing have become essential knowledge for all software developers. A software developer can expect to use the concepts presented in this book many times during his or her career. This chapter introduces the subject of software testing by describing the activities of a test engineer, defining a number of key terms, and then explaining the central notion of test coverage.

Software is a key ingredient in many of the devices and systems that pervade our society. Software defines the behavior of network routers, financial networks, telephone switching networks, the Web, and other infrastructure of modern life. Software is an essential component of embedded applications that control exotic applications such as airplanes, spaceships, and air traffic control systems, as well as mundane appliances such as watches, ovens, cars, DVD players, garage door openers, cell phones, and remote controllers. Modern households have over 50 processors, and some new cars have over 100; all of them running software that optimistic consumers assume will never fail! Although many factors affect the engineering of reliable software, including, of course, careful design and sound process management, testing is the primary method that the industry uses to evaluate software under development. Fortunately, a few basic software testing concepts can be used to design tests for a large variety of software applications. A goal of this book is to present these concepts in such a way that the student or practicing engineer can easily apply them to any software testing situation.

This textbook differs from other software testing books in several respects. The most important difference is in how it views testing techniques. In his landmark book *Software Testing Techniques*, Beizer wrote that testing is simple – all a tester needs to do is "find a graph and cover it." Thanks to Beizer's insight, it became evident to us that the myriad testing techniques present in the literature have much more in common than is obvious at first glance. Testing techniques typically are presented in the context of a particular software artifact (for example, a requirements document or code) or a particular phase of the lifecycle (for example, requirements analysis or implementation). Unfortunately, such a presentation obscures the underlying similarities between techniques. This book clarifies these similarities.

It turns out that graphs do not characterize all testing techniques well; other abstract models are necessary. Much to our surprise, we have found that a small number of abstract models suffice: graphs, logical expressions, input domain characterizations, and syntactic descriptions. The main contribution of this book is to simplify testing by classifying coverage criteria into these four categories, and this is why Part II of this book has exactly four chapters.

This book provides a balance of theory and practical application, thereby presenting testing as a collection of objective, quantitative activities that can be measured and repeated. The theory is based on the published literature, and presented without excessive formalism. Most importantly, the theoretical concepts are presented when needed to support the practical activities that test engineers follow. That is, this book is intended for software developers.

1.1 ACTIVITIES OF A TEST ENGINEER

In this book, a *test engineer* is an information technology (IT) professional who is in charge of one or more technical test activities, including designing test inputs, producing test case values, running test scripts, analyzing results, and reporting results to developers and managers. Although we cast the description in terms of test engineers, every engineer involved in software development should realize that he or she sometimes wears the hat of a test engineer. The reason is that each software artifact produced over the course of a product's development has, or should have, an associated set of test cases, and the person best positioned to define these test cases is often the designer of the artifact. A *test manager* is in charge of one or more test engineers. Test managers set test policies and processes, interact with other managers on the project, and otherwise help the engineers do their work.

Figure 1.1 shows some of the major activities of test engineers. A test engineer must design tests by creating test requirements. These requirements are then

Figure 1.1. Activities of test engineers.

transformed into actual values and scripts that are ready for execution. These executable tests are run against the software, denoted P in the figure, and the results are evaluated to determine if the tests reveal a fault in the software. These activities may be carried out by one person or by several, and the process is monitored by a test manager.

One of a test engineer's most powerful tools is a formal coverage criterion. Formal coverage criteria give test engineers ways to decide what test inputs to use during testing, making it more likely that the tester will find problems in the program and providing greater assurance that the software is of high quality and reliability. Coverage criteria also provide stopping rules for the test engineers. The technical core of this book presents the coverage criteria that are available, describes how they are supported by tools (commercial and otherwise), explains how they can best be applied, and suggests how they can be integrated into the overall development process.

Software testing activities have long been categorized into levels, and two kinds of levels have traditionally been used. The most often used level categorization is based on traditional software process steps. Although most types of tests can only be run after some part of the software is implemented, tests can be designed and constructed during all software development steps. The most time-consuming parts of testing are actually the test design and construction, so test activities can and should be carried out throughout development. The second-level categorization is based on the attitude and thinking of the testers.

1.1.1 Testing Levels Based on Software Activity

Tests can be derived from requirements and specifications, design artifacts, or the source code. A different level of testing accompanies each distinct software development activity:

- Acceptance Testing – assess software with respect to requirements.
- System Testing – assess software with respect to architectural design.
- Integration Testing – assess software with respect to subsystem design.
- Module Testing – assess software with respect to detailed design.
- Unit Testing – assess software with respect to implementation.

Figure 1.2 illustrates a typical scenario for testing levels and how they relate to software development activities by isolating each step. Information for each test level is typically derived from the associated development activity. Indeed, standard advice is to design the tests concurrently with each development activity, even though the software will not be in an executable form until the implementation phase. The reason for this advice is that the mere process of explicitly articulating tests can identify defects in design decisions that otherwise appear reasonable. Early identification of defects is by far the best means of reducing their ultimate cost. Note that this diagram is **not** intended to imply a waterfall process. The synthesis and analysis activities generically apply to any development process.

The *requirements analysis* phase of software development captures the customer's needs. *Acceptance testing* is designed to determine whether the completed software in fact meets these needs. In other words, acceptance testing probes

Figure 1.2. Software development activities and testing levels – the "V Model".

whether the software does what the users want. Acceptance testing must involve users or other individuals who have strong domain knowledge.

The *architectural design* phase of software development chooses components and connectors that together realize a system whose specification is intended to meet the previously identified requirements. *System testing* is designed to determine whether the assembled system meets its specifications. It assumes that the pieces work individually, and asks if the system works as a whole. This level of testing usually looks for design and specification problems. It is a very expensive place to find lower-level faults and is usually not done by the programmers, but by a separate testing team.

The *subsystem design* phase of software development specifies the structure and behavior of subsystems, each of which is intended to satisfy some function in the overall architecture. Often, the subsystems are adaptations of previously developed software. *Integration testing* is designed to assess whether the interfaces between modules (defined below) in a given subsystem have consistent assumptions and communicate correctly. Integration testing must assume that modules work correctly. Some testing literature uses the terms integration testing and system testing interchangeably; in this book, integration testing does **not** refer to testing the integrated system or subsystem. Integration testing is usually the responsibility of members of the development team.

The *detailed design* phase of software development determines the structure and behavior of individual modules. A program *unit*, or procedure, is one or more contiguous program statements, with a name that other parts of the software use to call it. Units are called functions in C and C++, procedures or functions in Ada, methods in Java, and subroutines in Fortran. A *module* is a collection of related units that are assembled in a file, package, or class. This corresponds to a file in C, a package in Ada, and a class in C++ and Java. *Module testing* is designed to assess individual modules in isolation, including how the component units interact with each other and their associated data structures. Most software development organizations make module testing the responsibility of the programmer.

Implementation is the phase of software development that actually produces code. *Unit testing* is designed to assess the units produced by the implementation phase and is the "lowest" level of testing. In some cases, such as when building general-purpose library modules, unit testing is done without knowledge of the encapsulating software application. As with module testing, most software development organizations make unit testing the responsibility of the programmer. It is straightforward to package unit tests together with the corresponding code through the use of tools such as *JUnit* for Java classes.

Not shown in Figure 1.2 is regression testing, a standard part of the maintenance phase of software development. *Regression testing* is testing that is done after changes are made to the software, and its purpose is to help ensure that the updated software still possesses the functionality it had before the updates.

Mistakes in requirements and high-level design wind up being implemented as faults in the program; thus testing can reveal them. Unfortunately, the software faults that come from requirements and design mistakes are visible only through testing months or years after the original mistake. The effects of the mistake tend to be dispersed throughout multiple software components; hence such faults are usually difficult to pin down and expensive to correct. On the positive side, even if tests cannot be executed, the very process of defining tests can identify a significant fraction of the mistakes in requirements and design. Hence, it is important for test planning to proceed concurrently with requirements analysis and design and not be put off until late in a project. Fortunately, through techniques such as use-case analysis, test planning is becoming better integrated with requirements analysis in standard software practice.

Although most of the literature emphasizes these levels in terms of **when** they are applied, a more important distinction is on the **types of faults** that we are looking for. The faults are based on the software **artifact** that we are testing, and the software **artifact** that we derive the tests from. For example, unit and module tests are derived to test units and modules, and we usually try to find faults that can be found when executing the units and modules individually.

One of the best examples of the differences between unit testing and system testing can be illustrated in the context of the infamous Pentium bug. In 1994, Intel introduced its Pentium microprocessor, and a few months later, Thomas Nicely, a mathematician at Lynchburg College in Virginia, found that the chip gave incorrect answers to certain floating-point division calculations.

The chip was slightly inaccurate for a few pairs of numbers; Intel claimed (probably correctly) that only one in nine billion division operations would exhibit reduced precision. The fault was the omission of five entries in a table of 1,066 values (part of the chip's circuitry) used by a division algorithm. The five entries should have contained the constant +2, but the entries were not initialized and contained zero instead. The MIT mathematician Edelman claimed that "the bug in the Pentium was an easy mistake to make, and a difficult one to catch," an analysis that misses one of the essential points. This was a very difficult mistake to find during system testing, and indeed, Intel claimed to have run millions of tests using this table. But the table entries were left empty because a loop termination condition was incorrect; that is, the loop stopped storing numbers before it was finished. This turns out to be a very simple fault to find during unit testing; indeed analysis showed that almost any unit level coverage criterion would have found this multimillion dollar mistake.

The Pentium bug not only illustrates the difference in testing levels, but it is also one of the best arguments for paying more attention to unit testing. There are no shortcuts – all aspects of software need to be tested.

On the other hand, some faults can only be found at the system level. One dramatic example was the launch failure of the first Ariane 5 rocket, which exploded 37 seconds after liftoff on June 4, 1996. The low-level cause was an unhandled floating-point conversion exception in an internal guidance system function. It turned out that the guidance system could never encounter the unhandled exception when used on the Ariane 4 rocket. In other words, the guidance system function is correct for Ariane 4. The developers of the Ariane 5 quite reasonably wanted to reuse the successful inertial guidance system from the Ariane 4, but no one reanalyzed the software in light of the substantially different flight trajectory of Ariane 5. Furthermore, the system tests that would have found the problem were technically difficult to execute, and so were not performed. The result was spectacular – and expensive!

Another public failure was the Mars lander of September 1999, which crashed due to a misunderstanding in the units of measure used by two modules created by separate software groups. One module computed thruster data in English units and forwarded the data to a module that expected data in metric units. This is a very typical integration fault (but in this case enormously expensive, both in terms of money and prestige).

One final note is that object-oriented (OO) software changes the testing levels. OO software blurs the distinction between units and modules, so the OO software testing literature has developed a slight variation of these levels. *Intramethod testing* is when tests are constructed for individual methods. *Intermethod testing* is when pairs of methods within the same class are tested in concert. *Intraclass testing* is when tests are constructed for a single entire class, usually as sequences of calls to methods within the class. Finally, *interclass testing* is when more than one class is tested at the same time. The first three are variations of unit and module testing, whereas interclass testing is a type of integration testing.

1.1.2 Beizer's Testing Levels Based on Test Process Maturity

Another categorization of levels is based on the test process maturity level of an organization. Each level is characterized by the goal of the test engineers. The following material is adapted from Beizer [29].

Level 0 There's no difference between testing and debugging.
Level 1 The purpose of testing is to show that the software works.
Level 2 The purpose of testing is to show that the software doesn't work.
Level 3 The purpose of testing is not to prove anything specific, but to reduce the risk of using the software.
Level 4 Testing is a mental discipline that helps all IT professionals develop higher quality software.

Level 0 is the view that testing is the same as debugging. This is the view that is naturally adopted by many undergraduate computer science majors. In most CS programming classes, the students get their programs to compile, then debug the programs with a few inputs chosen either arbitrarily or provided by the professor.

This model does not distinguish between a program's incorrect behavior and a mistake within the program, and does very little to help develop software that is reliable or safe.

In **Level 1** testing, the purpose is to show correctness. While a significant step up from the naive level 0, this has the unfortunate problem that in any but the most trivial of programs, correctness is virtually impossible to either achieve or demonstrate. Suppose we run a collection of tests and find no failures. What do we know? Should we assume that we have good software or just bad tests? Since the goal of correctness is impossible, test engineers usually have no strict goal, real stopping rule, or formal test technique. If a development manager asks how much testing remains to be done, the test manager has no way to answer the question. In fact, test managers are in a powerless position because they have no way to quantitatively express or evaluate their work.

In **Level 2** testing, the purpose is to show failures. Although looking for failures is certainly a valid goal, it is also a negative goal. Testers may enjoy finding the problem, but the developers never want to find problems – they want the software to work (level 1 thinking is natural for the developers). Thus, level 2 testing puts testers and developers into an adversarial relationship, which can be bad for team morale. Beyond that, when our primary goal is to look for failures, we are still left wondering what to do if no failures are found. Is our work done? Is our software very good, or is the testing weak? Having confidence in when testing is complete is an important goal for all testers.

The thinking that leads to **Level 3** testing starts with the realization that testing can show the presence, but not the absence, of failures. This lets us accept the fact that whenever we use software, we incur some risk. The risk may be small and the consequences unimportant, or the risk may be great and the consequences catastrophic, but risk is always there. This allows us to realize that the entire development team wants the same thing – to reduce the risk of using the software. In level 3 testing, both testers and developers work together to reduce risk.

Once the testers and developers are on the same "team," an organization can progress to real **Level 4** testing. Level 4 thinking defines testing as *a mental discipline that increases quality*. Various ways exist to increase quality, of which creating tests that cause the software to fail is only one. Adopting this mindset, test engineers can become the technical leaders of the project (as is common in many other engineering disciplines). They have the primary responsibility of measuring and improving software quality, and their expertise should help the developers. An analogy that Beizer used is that of a spell checker. We often think that the purpose of a spell checker is to find misspelled words, but in fact, the best purpose of a spell checker is to improve our ability to spell. Every time the spell checker finds an incorrectly spelled word, we have the opportunity to learn how to spell the word correctly. The spell checker is the "expert" on spelling quality. In the same way, level 4 testing means that the purpose of testing is to improve the ability of the developers to produce high quality software. The testers should train your developers.

As a reader of this book, you probably start at level 0, 1, or 2. Most software developers go through these levels at some stage in their careers. If you work in software development, you might pause to reflect on which testing level describes your company or team. The rest of this chapter should help you move to level 2 thinking, and to understand the importance of level 3. Subsequent chapters will give

you the knowledge, skills, and tools to be able to work at level 3. The ultimate goal of this book is to provide a philosophical basis that will allow readers to become "change agents" in their organizations for level 4 thinking, and test engineers to become **software quality experts**.

1.1.3 Automation of Test Activities

Software testing is expensive and labor intensive. Software testing requires up to 50% of software development costs, and even more for safety-critical applications. One of the goals of software testing is to automate as much as possible, thereby significantly reducing its cost, minimizing human error, and making regression testing easier.

Software engineers sometimes distinguish *revenue tasks*, which contribute directly to the solution of a problem, from *excise* tasks, which do not. For example, compiling a Java class is a classic excise task because, although necessary for the class to become executable, compilation contributes nothing to the particular behavior of that class. In contrast, determining which methods are appropriate to define a given data abstraction as a Java class is a revenue task. Excise tasks are candidates for automation; revenue tasks are not. Software testing probably has more excise tasks than any other aspect of software development. Maintaining test scripts, rerunning tests, and comparing expected results with actual results are all common excise tasks that routinely consume large chunks of test engineer's time. Automating excise tasks serves the test engineer in many ways. First, eliminating excise tasks eliminates drudgery, thereby making the test engineers job more satisfying. Second, automation frees up time to focus on the fun and challenging parts of testing, namely the revenue tasks. Third, automation can help eliminate errors of omission, such as failing to update all the relevant files with the new set of expected results. Fourth, automation eliminates some of the variance in test quality caused by differences in individual's abilities.

Many testing tasks that defied automation in the past are now candidates for such treatment due to advances in technology. For example, generating test cases that satisfy given test requirements is typically a hard problem that requires intervention from the test engineer. However, there are tools, both research and commercial, that automate this task to varying degrees.

EXERCISES

Section 1.1.

1. What are some of the factors that would help a development organization move from Beizer's testing level 2 (*testing is to show errors*) to testing level 4 (*a mental discipline that increases quality*)?
2. The following exercise is intended to encourage you to think of testing in a more rigorous way than you may be used to. The exercise also hints at the strong relationship between specification clarity, faults, and test cases.[1]
   (a) Write a Java method with the signature
   `public static Vector union (Vector a, Vector b)`
   The method should return a Vector of objects that are in either of the two argument Vectors.

(b) Upon reflection, you may discover a variety of defects and ambiguities in the given assignment. In other words, ample opportunities for faults exist. Identify as many possible faults as you can. (*Note:* `Vector` *is a Java* `Collection` *class. If you are using another language, interpret* `Vector` *as a list.*)

(c) Create a set of test cases that you think would have a reasonable chance of revealing the faults you identified above. Document a rationale for each test in your test set. If possible, characterize all of your rationales in some concise summary. Run your tests against your implementation.

(d) Rewrite the method signature to be precise enough to clarify the defects and ambiguities identified earlier. You might wish to illustrate your specification with examples drawn from your test cases.

1.2 SOFTWARE TESTING LIMITATIONS AND TERMINOLOGY

As said in the previous section, one of the most important limitations of software testing is that testing can show only the presence of failures, not their absence. This is a fundamental, theoretical limitation; generally speaking, the problem of finding all failures in a program is undecidable. Testers often call a successful (or effective) test one that finds an error. While this is an example of level 2 thinking, it is also a characterization that is often useful and that we will use later in this book.

The rest of this section presents a number of terms that are important in software testing and that will be used later in this book. Most of these are taken from standards documents, and although the phrasing is ours, we try to be consistent with the standards. Useful standards for reading in more detail are the IEEE Standard Glossary of Software Engineering Terminology, DOD-STD-2167A and MIL-STD-498 from the US Department of Defense, and the British Computer Society's Standard for Software Component Testing.

One of the most important distinctions to make is between validation and verification.

Definition 1.1 Validation: The process of evaluating software at the end of software development to ensure compliance with intended usage.

Definition 1.2 Verification: The process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase.

Verification is usually a more technical activity that uses knowledge about the individual software artifacts, requirements, and specifications. Validation usually depends on domain knowledge; that is, knowledge of the application for which the software is written. For example, validation of software for an airplane requires knowledge from aerospace engineers and pilots.

The acronym "IV&V" stands for "independent verification and validation," where "independent" means that the evaluation is done by nondevelopers. Sometimes the IV&V team is within the same project, sometimes the same company, and sometimes it is entirely an external entity. In part because of the independent nature of IV&V, the process often is not started until the software is complete and is often done by people whose expertise is in the application domain rather than software

development. This can sometimes mean that validation is given more weight than verification.

Two terms that we have already used are fault and failure. Understanding this distinction is the first step in moving from level 0 thinking to level 1 thinking. We adopt the definition of software fault, error, and failure from the dependability community.

Definition 1.3 Software Fault: A static defect in the software.

Definition 1.4 Software Error: An incorrect internal state that is the manifestation of some fault.

Definition 1.5 Software Failure: External, incorrect behavior with respect to the requirements or other description of the expected behavior.

Consider a medical doctor making a diagnosis for a patient. The patient enters the doctor's office with a list of *failures* (that is, *symptoms*). The doctor then must discover the *fault*, or root cause of the symptom. To aid in the diagnosis, a doctor may order tests that look for anomalous internal conditions, such as high blood pressure, an irregular heartbeat, high levels of blood glucose, or high cholesterol. In our terminology, these anomalous internal conditions correspond to *errors*.

While this analogy may help the student clarify his or her thinking about faults, errors, and failures, software testing and a doctor's diagnosis differ in one crucial way. Specifically, faults in software are *design mistakes*. They do not appear spontaneously, but rather exist as a result of some (unfortunate) decision by a human. Medical problems (as well as faults in computer system hardware), on the other hand, are often a result of physical degradation. This distinction is important because it explains the limits on the extent to which any process can hope to control software faults. Specifically, since no foolproof way exists to catch arbitrary mistakes made by humans, we cannot eliminate all faults from software. In colloquial terms, we can make software development foolproof, but we cannot, and should not attempt to, make it damn-foolproof.

For a more technical example of the definitions of fault, error, and failure, consider the following Java method:

```
public static int numZero (int[] x) {
  // Effects: if x == null throw NullPointerException
  //    else return the number of occurrences of 0 in x
  int count = 0;
  for (int i = 1; i < x.length; i++)
  {
    if (x[i] == 0)
    {
      count++;
    }
  }
  return count;
}
```

The fault in this program is that it starts looking for zeroes at index 1 instead of index 0, as is necessary for arrays in Java. For example, `numZero ([2, 7, 0])` correctly evaluates to 1, while `numZero ([0, 7, 2])` incorrectly evaluates to 0. In both of these cases the fault is executed. Although both of these cases result in an error, only the second case results in failure. To understand the error states, we need to identify the state for the program. The state for `numZero` consists of values for the variables `x`, `count`, `i`, and the program counter (denoted `PC`). For the first example given above, the state at the if statement on the very first iteration of the loop is ($\mathtt{x} = [2, 7, 0]$, $\mathtt{count} = 0$, $\mathtt{i} = 1$, $\mathtt{PC} = \mathtt{if}$). Notice that this state is in error precisely because the value of `i` should be zero on the first iteration. However, since the value of `count` is coincidentally correct, the error state does not propagate to the output, and hence the software does not fail. In other words, a state is in error simply if it is not the expected state, even if all of the values in the state, considered in isolation, are acceptable. More generally, if the required sequence of states is $s_0, s_1, s_2, \ldots$, and the actual sequence of states is $s_0, s_2, s_3, \ldots$, then state s_2 is in error in the second sequence.

In the second case the corresponding (error) state is ($\mathtt{x} = [0, 7, 2]$, $\mathtt{count} = 0$, $\mathtt{i} = 1$, $\mathtt{PC} = \mathtt{if}$). In this case, the error propagates to the variable `count` and is present in the return value of the method. Hence a failure results.

The definitions of fault and failure allow us to distinguish testing from debugging.

Definition 1.6 Testing: Evaluating software by observing its execution.

Definition 1.7 Test Failure: Execution that results in a failure.

Definition 1.8 Debugging: The process of finding a fault given a failure.

Of course the central issue is that for a given fault, not all inputs will "trigger" the fault into creating incorrect output (a failure). Also, it is often very difficult to relate a failure to the associated fault. Analyzing these ideas leads to the fault/failure model, which states that three conditions must be present for a failure to be observed.

1. The location or locations in the program that contain the fault must be reached (*Reachability*).
2. After executing the location, the state of the program must be incorrect (*Infection*).
3. The infected state must propagate to cause some output of the program to be incorrect (*Propagation*).

This "RIP" model is very important for coverage criteria such as mutation (Chapter 5) and for automatic test data generation. It is important to note that the RIP model applies even in the case of faults of omission. In particular, when execution traverses the missing code, the program counter, which is part of the internal state, necessarily has the wrong value.

The next definitions are less standardized and the literature varies widely. The definitions are our own but are consistent with common usage. A test engineer must recognize that tests include more than just input values, but are actually multipart

software artifacts. The piece of a test case that is referred to the most often is what we call the test case value.

Definition 1.9 Test Case Values: The input values necessary to complete some execution of the software under test.

Note that the definition of test case values is quite broad. In a traditional batch environment, the definition is extremely clear. In a Web application, a complete execution might be as small as the generation of part of a simple Web page, or it might be as complex as the completion of a set of commercial transactions. In a real-time system such as an avionics application, a complete execution might be a single frame, or it might be an entire flight.

Test case values are the inputs to the program that test engineers typically focus on during testing. They really define what sort of testing we will achieve. However, test case values are not enough. In addition to test case values, other inputs are often needed to run a test. These inputs may depend on the source of the tests, and may be commands, user inputs, or a software method to call with values for its parameters. In order to evaluate the results of a test, we must know what output a correct version of the program would produce for that test.

Definition 1.10 Expected Results: The result that will be produced when executing the test if and only if the program satisfies its intended behavior.

Two common practical problems associated with software testing are how to provide the right values to the software and observing details of the software's behavior. These two ideas are used to refine the definition of a test case.

Definition 1.11 Software Observability: How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment, and other hardware and software components.

Definition 1.12 Software Controllability: How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors.

These ideas are easily illustrated in the context of embedded software. Embedded software often does not produce output for human consumption, but affects the behavior of some piece of hardware. Thus, observability will be quite low. Likewise, software for which all inputs are values entered from a keyboard is easy to control. But an embedded program that gets its inputs from hardware sensors is more difficult to control and some inputs may be difficult, dangerous or impossible to supply (for example, how does the automatic pilot behave when a train jumps off-track). Many observability and controllability problems can be addressed with simulation, by extra software built to "bypass" the hardware or software components that interfere with testing. Other applications that sometimes have low observability and controllability include component-based software, distributed software and Web applications.

Depending on the software, the level of testing, and the source of the tests, the tester may need to supply other inputs to the software to affect controllability or observability. For example, if we are testing software for a mobile telephone, the test case values may be long distance phone numbers. We may also need to turn the

phone on to put it in the appropriate state and then we may need to press "talk" and "end" buttons to view the results of the test case values and terminate the test. These ideas are formalized as follows.

Definition 1.13 Prefix Values: Any inputs necessary to put the software into the appropriate state to receive the test case values.

Definition 1.14 Postfix Values: Any inputs that need to be sent to the software after the test case values are sent.

Postfix values can be subdivided into two types.

Definition 1.15 Verification Values: Values necessary to see the results of the test case values.

Definition 1.16 Exit Commands: Values needed to terminate the program or otherwise return it to a stable state.

A test case is the combination of all these components (test case values, expected results, prefix values, and postfix values). When it is clear from context, however, we will follow tradition and use the term "test case" in place of "test case values."

Definition 1.17 Test Case: A test case is composed of the test case values, expected results, prefix values, and postfix values necessary for a complete execution and evaluation of the software under test.

We provide an explicit definition for a test set to emphasize that coverage is a property of a set of test cases, rather than a property of a single test case.

Definition 1.18 Test Set: A test set is simply a set of test cases.

Finally, wise test engineers automate as many test activities as possible. A crucial way to automate testing is to prepare the test inputs as executable tests for the software. This may be done as Unix shell scripts, input files, or through the use of a tool that can control the software or software component being tested. Ideally, the execution should be complete in the sense of running the software with the test case values, getting the results, comparing the results with the expected results, and preparing a clear report for the test engineer.

Definition 1.19 Executable Test Script: A test case that is prepared in a form to be executed automatically on the test software and produce a report.

The only time a test engineer would not want to automate is if the cost of automation outweighs the benefits. For example, this may happen if we are sure the test will only be used once or if the automation requires knowledge or skills that the test engineer does not have.

EXERCISES

Section 1.2.

1. For what do testers use automation? What are the limitations of automation?
2. How are faults and failures related to testing and debugging?

3. Below are four faulty programs. Each includes a test case that results in failure. Answer the following questions about each program.

```
public int findLast (int[] x, int y) {
//Effects: If x==null throw NullPointerException
//   else return the index of the last element
//   in x that equals y.
//   If no such element exists, return -1
  for (int i=x.length-1; i > 0; i--)
  {
    if (x[i] == y)
    {
      return i;
    }
  }
  return -1;
}
  // test:  x=[2, 3, 5]; y = 2
  //        Expected = 0
```

```
public static int lastZero (int[] x) {
//Effects: if x==null throw NullPointerException
//   else return the index of the LAST 0 in x.
//   Return -1 if 0 does not occur in x

  for (int i = 0; i < x.length; i++)
  {
    if (x[i] == 0)
    {
      return i;
    }
  }
  return -1;
}
  // test:  x=[0, 1, 0]
  //        Expected = 2
```

```
public int countPositive (int[] x) {
//Effects: If x==null throw NullPointerException
//   else return the number of
//        positive elements in x.
  int count = 0;
  for (int i=0; i < x.length; i++)
  {
    if (x[i] >= 0)
    {
      count++;
    }
  }
  return count;
}
  // test:  x=[-4, 2, 0, 2]
  //        Expected = 2
```

```
public static int oddOrPos(int[] x) {
//Effects: if x==null throw NullPointerException
// else return the number of elements in x that
//      are either odd or positive (or both)
  int count = 0;
  for (int i = 0; i < x.length; i++)
  {
    if (x[i]% 2 == 1 || x[i] > 0)
    {
      count++;
    }
  }
  return count;
}
  // test:  x=[-3, -2, 0, 1, 4]
  //        Expected = 3
```

(a) Identify the fault.
(b) If possible, identify a test case that does **not** execute the fault.
(c) If possible, identify a test case that executes the fault, but does **not** result in an error state.
(d) If possible identify a test case that results in an error, but **not** a failure. Hint: Don't forget about the program counter.
(e) For the given test case, identify the first error state. Be sure to describe the complete state.
(f) Fix the fault and verify that the given test now produces the expected output.

1.3 COVERAGE CRITERIA FOR TESTING

Some ill-defined terms occasionally used in testing are "complete testing," "exhaustive testing," and "full coverage." These terms are poorly defined because of a fundamental theoretical limitation of software. Specifically, the number of potential inputs for most programs is so large as to be effectively infinite. Consider a Java compiler – the number of potential inputs to the compiler is not just all Java programs, or even all almost correct Java programs, but all strings. The only limitation is the size of the file that can be read by the parser. Therefore, the number of inputs is effectively infinite and cannot be explicitly enumerated.

This is where formal coverage criteria come in. Since we cannot test with all inputs, coverage criteria are used to decide which test inputs to use. The software testing community believes that effective use of coverage criteria makes it more likely that test engineers will find faults in a program and provides informal assurance that the software is of high quality and reliability. While this is, perhaps, more an article of faith than a scientifically supported proposition, it is, in our view, the best option currently available. From a practical perspective, coverage criteria provide useful rules for when to stop testing.

This book defines coverage criteria in terms of test requirements. The basic idea is that we want our set of test cases to have various properties, each of which is provided (or not) by an individual test case.[2]

Definition 1.20 Test Requirement: A test requirement is a specific element of a software artifact that a test case must satisfy or cover.

Test requirements usually come in sets, and we use the abbreviation *TR* to denote a set of test requirements.

Test requirements can be described with respect to a variety of software artifacts, including the source code, design components, specification modeling elements, or even descriptions of the input space. Later in this book, test requirements will be generated from all of these.

Let's begin with a non-software example. Suppose we are given the enviable task of testing bags of jelly beans. We need to come up with ways to sample from the bags. Suppose these jelly beans have the following six flavors and come in four colors: Lemon (colored Yellow), Pistachio (Green), Cantaloupe (Orange), Pear (White), Tangerine (also Orange), and Apricot (also Yellow). A simple approach to testing might be to test one jelly bean of each flavor. Then we have six test requirements, one for each flavor. We satisfy the test requirement "Lemon" by selecting and, of course, tasting a Lemon jelly bean from a bag of jelly beans. The reader might wish to ponder how to decide, prior to the tasting step, if a given Yellow jelly bean is Lemon or Apricot. This dilemma illustrates a classic controllability issue.

As a more software-oriented example, if the goal is to cover all decisions in the program (branch coverage), then each decision leads to two test requirements, one for the decision to evaluate to false, and one for the decision to evaluate to true. If every method must be called at least once (call coverage), each method leads to one test requirement.

A coverage criterion is simply a recipe for generating test requirements in a systematic way:

Definition 1.21 Coverage Criterion: A coverage criterion is a rule or collection of rules that impose test requirements on a test set.

That is, the criterion describes the test requirements in a complete and unambiguous manner. The "flavor criterion" yields a simple strategy for selecting jelly beans. In this case, the set of test requirements, TR, can be formally written out as

$$TR = \{flavor = Lemon,\ flavor = Pistachio,\ flavor = Cantaloupe,\\ flavor = Pear,\ flavor = Tangerine,\ flavor = Apricot\}$$

Test engineers need to know how good a collection of tests is, so we measure test sets against a criterion in terms of coverage.

Definition 1.22 Coverage: Given a set of test requirements TR for a coverage criterion C, a test set T satisfies C if and only if for every test requirement tr in TR, at least one test t in T exists such that t satisfies tr.

To continue the example, a test set T with 12 beans: three Lemon, one Pistachio, two Cantaloupe, one Pear, one Tangerine, and four Apricot satisfies the "flavor criterion." Notice that it is perfectly acceptable to satisfy a given test requirement with more than one test.

Coverage is important for two reasons. First, it is sometimes expensive to satisfy a coverage criterion, so we want to compromise by trying to achieve a certain coverage level.

Definition 1.23 Coverage Level: Given a set of test requirements TR and a test set T, the coverage level is simply the ratio of the number of test requirements satisfied by T to the size of TR.

Second, and more importantly, some requirements cannot be satisfied. Suppose Tangerine jelly beans are rare, some bags may not contain any, or it may simply be too difficult to find a Tangerine bean. In this case, the flavor criterion cannot be 100% satisfied, and the maximum coverage level possible is 5/6 or 83%. It often makes sense to drop unsatisfiable test requirements from the set TR – or to replace them with less stringent test requirements.

Test requirements that cannot be satisfied are called *infeasible*. Formally, no test case values exist that meet the test requirements. Examples for specific software criteria will be shown throughout the book, but some may already be familiar. Dead code results in infeasible test requirements because the statements cannot be reached. The detection of infeasible test requirements is formally undecidable for most coverage criteria, and even though some researchers have tried to find partial solutions, they have had only limited success. Thus, 100% coverage is impossible in practice.

Coverage criteria are traditionally used in one of two ways. One method is to directly generate test case values to satisfy a given criterion. This method is often assumed by the research community and is the most obvious way to use criteria. It is also very hard in some cases, particularly if we do not have enough automated tools to support test case value generation. The other method is to generate test case values externally (by hand or using a pseudo-random tool, for example) and then measure the tests against the criterion in terms of their coverage. This method is usually favored by industry practitioners, because generating tests to directly satisfy the criterion is too hard. Unfortunately, this use is sometimes misleading. If our tests do not reach 100% coverage, what does that mean? We really have no data on how much, say, 99% coverage is worse than 100% coverage, or 90%, or even 75%. Because of this use of the criteria to evaluate existing test sets, coverage criteria are sometimes called *metrics*.

This distinction actually has a strong theoretical basis. A *generator* is a procedure that automatically generates values to satisfy a criterion, and a *recognizer* is a

procedure that decides whether a given set of test case values satisfies a criterion. Theoretically, both problems are provably undecidable in the general case for most criteria. In practice, however, it is possible to recognize whether test cases satisfy a criterion far more often than it is possible to generate tests that satisfy the criterion. The primary problem with recognition is infeasible test requirements; if no infeasible test requirements are present then the problem becomes decidable.

In practical terms of commercial automated test tools, a generator corresponds to a tool that automatically creates test case values. A recognizer is a coverage analysis tool. Coverage analysis tools are quite plentiful, both as commercial products and freeware.

It is important to appreciate that the set TR depends on the specific artifact under test. In the jelly bean example, the test requirement *color* = *Purple* doesn't make sense because we assumed that the factory does not make Purple jelly beans. In the software context, consider statement coverage. The test requirement "Execute statement 42" makes sense only if the program under test does indeed have a statement 42. A good way to think of this issue is that the test engineer starts with a given software artifact and then chooses a particular coverage criterion. Combining the artifact with the criterion yields the specific set TR that is relevant to the test engineer's task.

Coverage criteria are often related to one another, and compared in terms of subsumption. Recall that the "flavor criterion" requires that every flavor be tried once. We could also define a "color criterion," which requires that we try one jelly bean of each color {*yellow, green, orange, white*}. If we satisfy the flavor criterion, then we have also implicitly satisfied the color criterion. This is the essence of subsumption; that satisfying one criterion will guarantee that another one is satisfied.

Definition 1.24 Criteria Subsumption: A coverage criterion C_1 subsumes C_2 if and only if every test set that satisfies criterion C_1 also satisfies C_2.

Note that this has to be true for **every** test set, not just some sets. Subsumption has a strong similarity with set subset relationships, but it is not exactly the same. Generally, a criterion C_1 can subsume another C_2 in one of two ways. The simpler way is if the test requirements for C_1 always form a superset of the requirements for C_2. For example, another jelly bean criterion may be to try all flavors whose name begins with the letter 'C'. This would result in the test requirements {*Cantaloupe*}, which is a subset of the requirements for the flavor criterion: {*Lemon, Pistachio, Cantaloupe, Pear, Tangerine, Apricot*}. Thus, the flavor criterion subsumes the "starts-with-C" criterion.

The relationship between the flavor and the color criteria illustrate the other way that subsumption can be shown. Since every flavor has a specific color, and every color is represented by at least one flavor, if we satisfy the flavor criterion we will also satisfy the color criterion. Formally, a many-to-one mapping exists between the requirements for the flavor criterion and the requirements for the color criterion. Thus, the flavor criterion subsumes the color criterion. (If a one-to-one mapping exists between requirements from two criteria, then they would subsume each other.)

For a more realistic software-oriented example, consider branch and statement coverage. (These should already be familiar, at least intuitively, and will be defined

formally in Chapter 2.) If a test set has covered every branch in a program (satisfied branch coverage), then the test set is guaranteed to have covered every statement as well. Thus, the branch coverage criterion subsumes the statement coverage criterion. We will return to subsumption with more rigor and more examples in subsequent chapters.

1.3.1 Infeasibility and Subsumption

A subtle relationship exists between infeasibility and subsumption. Specifically, sometimes a criterion C_1 will subsume another criterion C_2 if and only if all test requirements are feasible. If some test requirements in C_1 are infeasible, however, C_1 may not subsume C_2.

Infeasible test requirements are common and occur quite naturally. Suppose we partition the jelly beans into *Fruits* and *Nuts*.[3] Now, consider the *Interaction Criterion*, where each flavor of bean is sampled in conjunction with some other flavor in the same block. Such a criterion has a useful counterpart in the software domain in cases where feature interactions are a source of concern. So, for example, we might try Lemon with Pear or Tangerine, but we would not try Lemon with itself or with Pistachio. We might think that the Interaction Criterion subsumes the Flavor criterion, since every flavor is tried in conjunction with some other flavor. Unfortunately, in our example, Pistachio is the only member of the *Nuts* block, and hence the test requirement to try it with some other flavor in the *Nuts* block is infeasible.

One possible strategy to reestablish subsumption is to replace each infeasible test requirement for the Interaction Criterion with the corresponding one from the Flavor criterion. In this example, we would simply taste Pistachio nuts by themselves. In general, it is desirable to define coverage criteria so that they are robust with respect to subsumption in the face of infeasible test requirements. This is not commonly done in the testing literature, but we make an effort to do so in this book.

That said, this problem is mainly theoretical and should not overly concern practical testers. Theoretically, sometimes a coverage criterion C_1 will subsume another C_2 if we assume that C_1 has **no** infeasible test requirements, but if C_1 does create an infeasible test requirement for a program, a test suite that satisfies C_1 while skipping the infeasible test requirements might also "skip" some test requirements from C_2 that are satisfiable. In practice, only a few test requirements for C_1 are infeasible for any given program, and if some are, it is often true that corresponding test requirements in C_2 will also be infeasible. If not, the few test cases that are lost will **probably** not make a difference in the test results.

1.3.2 Characteristics of a Good Coverage Criterion

Given the above discussion, an interesting question is "what makes a coverage criterion good?" Certainly, no definitive answers exist to this question, a fact that may partly explain why so many coverage criteria have been designed. However, three important issues can affect the use of coverage criteria.

1. The difficulty of computing test requirements
2. The difficulty of generating tests
3. How well the tests reveal faults

Subsumption is at best a very rough way to compare criteria. Our intuition may tell us that if one criterion subsumes another, then it should reveal more faults. However, no theoretical guarantee exists and the experimental studies have usually not been convincing and are far from complete. Nevertheless, the research community has reasonably wide agreement on relationships among some criteria. The difficulty of computing test requirements will depend on the artifact being used as well as the criterion. The fact that the difficulty of generating tests can be directly related to how well the tests reveal faults should not be surprising. A software tester must strive for balance and choose criteria that have the right cost / benefit tradeoffs for the software under test.

EXERCISES

Section 1.3.

1. Suppose that coverage criterion C_1 subsumes coverage criterion C_2. Further suppose that test set T_1 satisfies C_1 and on program P test set T_2 satisfies C_2, also on P.
   (a) Does T_1 necessarily satisfy C_2? Explain.
   (b) Does T_2 necessarily satisfy C_1? Explain.
   (c) If P contains a fault, and T_2 reveals the fault, T_1 does **not** necessarily also reveal the fault. Explain.[4]
2. How else could we compare test criteria besides subsumption?

1.4 OLDER SOFTWARE TESTING TERMINOLOGY

The testing research community has been very active in the past two decades, and some of our fundamental views of what and how to test have changed. This section presents some of the terminology that has been in use for many years, but for various reasons has become dated. Despite the fact that they are not as relevant now as they were at one time, these terms are still used and it is important that testing students and professionals be familiar with them.

From an abstract perspective, black-box and white-box testing are very similar. In this book in particular, we present testing as proceeding from abstract models of the software such as graphs, which can as easily be derived from a black-box view or a white-box view. Thus, one of the most obvious effects of the unique philosophical structure of this book is that these two terms become obsolete.

Definition 1.25 Black-box testing: Deriving tests from external descriptions of the software, including specifications, requirements, and design.

Definition 1.26 White-box testing: Deriving tests from the source code internals of the software, specifically including branches, individual conditions, and statements.

In the early 1980s, a discussion took place over whether testing should proceed from the top down or from the bottom up. This was an echo of a previous discussion over how to develop software. This distinction has pretty much disappeared as we

first learned that top-down testing is impractical, then OO design pretty much made the distinction obsolete. The following pair of definitions assumes that software can be viewed as a tree of software procedures, where the edges represent calls and the root of the tree is the main procedure.

Definition 1.27 Top-Down Testing: Test the main procedure, then go down through procedures it calls, and so on.

Definition 1.28 Bottom-Up Testing: Test the leaves in the tree (procedures that make no calls), and move up to the root. Each procedure is tested only if all of its children have been tested.

OO software leads to a more general problem. The relationships among classes can be formulated as general graphs with cycles, requiring test engineers to make the difficult choice of what order to test the classes in. This problem is discussed in Chapter 6.

Some parts of the literature separate static and dynamic testing as follows:

Definition 1.29 Static Testing: Testing without executing the program. This includes software inspections and some forms of analysis.

Definition 1.30 Dynamic Testing: Testing by executing the program with real inputs.

Most of the literature currently uses "testing" to refer to dynamic testing and "static testing" is called "verification activities." We follow that use in this book and it should be pointed out that this book is only concerned with dynamic or *execution-based* testing.

One last term bears mentioning because of the lack of definition. *Test Strategy* has been used to mean a variety of things, including coverage criterion, test process, and technologies used. We will avoid using it.

1.5 BIBLIOGRAPHIC NOTES

All books on software testing and all researchers owe major thanks to the landmark books in 1979 by Myers [249], in 1990 by Beizer [29], and in 2000 by Binder [33]. Some excellent overviews of unit testing criteria have also been published, including one by White [349] and more recently by Zhu, Hall, and May [367]. The statement that software testing requires up to 50 percent of software development costs is from Myers and Sommerville [249, 316]. The recent text from Pezze and Young [289] reports relevant processes, principles, and techniques from the testing literature, and includes many useful classroom materials. The Pezze and Young text presents coverage criteria in the traditional lifecycle-based manner, and does not organize criteria into the four abstract models discussed in this chapter.

Numerous other software testing books were not intended as textbooks, or do not offer general coverage for classroom use. Beizer's *Software System Testing and Quality Assurance* [28] and Hetzel's *The Complete Guide to Software Testing* [160] cover various aspects of management and process for software testing. Several books cover specific aspects of testing [169, 227, 301]. The STEP project at Georgia

Institute of Technology resulted in a comprehensive survey of the practice of software testing by Department of Defense contractors in the 1980s [100].

The definition of *unit* is from Stevens, Myers and Constantine [318], and the definition of *module* is from Sommerville [316]. The definition of *integration testing* is from Beizer [29]. The clarification for OO testing levels with the terms *intra-method*, *inter-method*, and *intra-class* testing is from Harrold and Rothermel [152] and *inter-class* testing is from Gallagher, Offutt and Cincotta [132].

The information for the Pentium bug and Mars lander was taken from several sources, including by Edelman, Moler, Nuseibeh, Knutson, and Peterson [111, 189, 244, 259, 286]. The accident report [209] is the best source for understanding the details of the Ariane 5 Flight 501 Failure.

The testing levels in Section 1.1.2 were first defined by Beizer [29].

The elementary result that finding all failures in a program is undecidable is due to Howden [165].

Most of the terminology in testing is from standards documents, including the IEEE Standard Glossary of Software Engineering Terminology [175], the US Department of Defense [260, 261], the US Federal Aviation Administration FAA-DO178B, and the British Computer Society's Standard for Software Component Testing [317]. The definitions for observability and controllability come from Freedman [129]. Similar definitions were also given in Binder's book *Testing Object-Oriented Systems* [33].

The fault/failure model was developed independently by Offutt and Morell in their dissertations [101, 246, 247, 262]. Morell used the terms execution, infection, and propagation [247, 246], and Offutt used reachability, sufficiency, and necessity [101, 262]. This book merges the two sets of terms by using what we consider to be the most descriptive terms.

The multiple parts of the test case that we use are based on research in test case specifications [23, 319].

One of the first discussions of infeasibility from other than a purely theoretical view was by Frankl and Weyuker [128]. The problem was shown to be undecidable by Goldberg et al. [136] and DeMillo and Offutt [101]. Some partial solutions have been presented [132, 136, 177, 273].

Budd and Angluin [51] analyzed the theoretical distinctions between generators and recognizers from a testing viewpoint. They showed that both problems are formally undecidable, and discussed tradeoffs in approximating the two.

Subsumption has been widely used as a way to analytically compare testing techniques. We follow Weiss [340] and Frankl and Weyuker [128] for our definition of subsumption. Frankl and Weyuker actually used the term *includes*. The term subsumption was defined by Clarke et al.: A criterion C_1 *subsumes* a criterion C_2 if and only if every set of execution paths P that satisfies C_1 also satisfies C_2 [81]. The term *subsumption* is currently the more widely used and the two definitions are equivalent; this book follows Weiss's suggestion to use the term *subsumes* to refer to Frankl and Weyuker's definition.

The descriptions of excise and revenue tasks were taken from Cooper [89].

Although this book does not focus heavily on the theoretical underpinnings of software testing, students interested in research should study such topics more in depth. A number of the papers are quite old and often do not appear in current

literature, and their ideas are beginning to disappear. The authors encourage the study of the older papers. Among those are truly seminal papers in the 1970s by Goodenough and Gerhart [138] and Howden [165], and Demillo, Lipton, Sayward, and Perlis [98, 99]. These papers were followed up and refined by Weyuker and Ostrand [343], Hamlet [147], Budd and Angluin [51], Gourlay [139], Prather [293], Howden [168], and Cherniavsky and Smith [67]. Later theoretical papers were contributed by Morell [247], Zhu [366], and Wah [335, 336]. Every PhD student's adviser will certainly have his or her own favorite theoretical papers, but this list should provide a good starting point.

NOTES

1 Liskov's *Program Development in Java*, especially chapters 9 and 10, is a great source for students who wish to pursue this direction further.
2 While this is a good general rule, exceptions exist. For example, test requirements for some logic coverage criteria demand pairs of related test cases instead of individual test cases.
3 The reader might wonder whether we need an *Other* category to ensure that we have a partition. In our example, we are ok, but in general, one would need such a category to handle jelly beans such as Potato, Spinach, or Ear Wax.
4 Correctly answering this question goes a long way towards understanding the weakness of the subsumption relation.

PART 2

Coverage Criteria

2

Graph Coverage

This chapter introduces the major test coverage criteria in use today. It starts out in a very theoretical way, but a firm grasp of the theoretical aspects of graphs and graph coverage makes the remainder of the chapter simpler. We first emphasize a generic view of a graph without regard to the graph's source. After this model is established, the rest of the chapter turns to practical applications by demonstrating how graphs can be obtained from various software artifacts and how the generic versions of the criteria are adapted to those graphs.

2.1 OVERVIEW

Directed graphs form the foundation for many coverage criteria. Given an artifact under test, the idea is to obtain a graph abstraction of that artifact. For example, the most common graph abstraction for source code maps code to a control flow graph. It is important to understand that the graph is not the same as the artifact, and that, indeed, artifacts typically have several useful, but nonetheless quite different, graph abstractions. The same abstraction that produces the graph from the artifact also maps test cases for the artifact to paths in the graph. Accordingly, a graph-based coverage criterion evaluates a test set for an artifact in terms of how the paths corresponding to the test cases "cover" the artifact's graph abstraction.

We give our basic notion of a graph below and will add additional structures later in the chapter when needed. A graph G formally is

- a set N of *nodes*
- a set N_0 of *initial nodes*, where $N_0 \subseteq N$
- a set N_f of *final nodes*, where $N_f \subseteq N$
- a set E of *edges*, where E is a subset of $N \times N$

For a graph to be useful for generating tests, it is necessary for N, N_0, and N_f to contain at least one node each. Sometimes, it is helpful to consider only part of a graph. A *subgraph* of a graph is also a graph and is defined by a subset of N, along with the corresponding subsets of N_0, N_f, and E. Specifically, if N_{sub} is a subset of

Figure 2.1. Graph (a) has a single initial node, graph (b) multiple initial nodes, and graph (c) (rejected) with no initial nodes.

N, then for the subgraph defined by N_{sub}, the set of initial nodes is $N_{sub} \cap N_0$, the set of final nodes is $N_{sub} \cap N_f$, and the set of edges is $(N_{sub} \times N_{sub}) \cap E$.

Note that more than one initial node can be present; that is, N_0 is a set. Having multiple initial nodes is necessary for some software artifacts, for example, if a class has multiple entry points, but sometimes we will restrict the graph to having one initial node. Edges are considered to be *from* one node and *to* another and written as (n_i, n_j). The edge's initial node n_i is sometimes called the *predecessor* and n_j is called the *successor*.

We always identify final nodes, and there must be at least one final node. The reason is that every test must start in some initial node and end in some final node. The concept of a final node depends on the kind of software artifact the graph represents. Some test criteria require tests to end in a particular final node. Other test criteria are satisfied with any node for a final node, in which case the set N_f is the same as the set N.

The term *node* has various synonyms. Graph theory texts sometimes call a node a *vertex*, and testing texts typically identify a node with the structure it represents, often a statement or a basic block. Similarly, graph theory texts sometimes call an edge an *arc*, and testing texts typically identify an edge with the structure it represents, often a branch. This section discusses graph criteria in a generic way; thus we stick to general graph terms.

Graphs are often drawn with bubbles and arrows. Figure 2.1 shows three example graphs. The nodes with incoming edges but no predecessor nodes are the initial nodes. The nodes with heavy borders are final nodes. Figure 2.1(a) has a single initial node and no cycles. Figure 2.1(b) has three initial nodes, as well as a cycle $([n_1, n_4, n_8, n_5, n_1])$. Figure 2.1(c) has no initial nodes, and so is not useful for generating test cases.

Path Examples	
1	n_0, n_3, n_7
2	n_1, n_4, n_8, n_5, n_1
3	n_2, n_6, n_9

Invalid Path Examples	
1	n_0, n_7
2	n_3, n_4
3	n_2, n_6, n_8

(a) Path examples

Reachability Examples	
1	*reach* $(n_0) = N - \{n_2, n_6\}$
2	*reach* $(n_0, n_1, n_2) = N$
3	*reach* $(n_4) = \{n_1, n_4, n_5, n_7, n_8, n_9\}$
4	*reach* $([n_6, n_9]) = \{ n_9 \}$

(b) Reachability examples

Figure 2.2. Example of paths.

A *path* is a sequence $[n_1, n_2, \ldots, n_M]$ of nodes, where each pair of adjacent nodes, (n_i, n_{i+1}), $1 \leq i < M$, is in the set E of edges. The length of a path is defined as the number of edges it contains. We sometimes consider paths and subpaths of length zero. A *subpath* of a path p is a subsequence of p (possibly p itself). Following the notation for edges, we say a path is *from* the first node in the path and *to* the last node in the path. It is also useful to be able to say that a path is *from* (or *to*) an edge e, which simply means that e is the first (or last) edge in the path.

Figure 2.2 shows a graph along with several example paths, and several examples that are not paths. For instance, the sequence $[n_0, n_7]$ is not a path because the two nodes are not connected by an edge.

Many test criteria require inputs that start at one node and end at another. This is only possible if those nodes are connected by a path. When we apply these criteria on specific graphs, we sometimes find that we have asked for a path that for some reason cannot be executed. For example, a path may demand that a loop be executed zero times in a situation where the program always executes the loop at least once. This kind of problem is based on the **semantics** of the software artifact that the graph represents. For now, we emphasize that we are looking only at the **syntax** of the graph.

We say that a node n (or an edge e) is *syntactically reachable* from node n_i if there exists a path from node n_i to n (or edge e). A node n (or edge e) is also *semantically reachable* if it is possible to execute at least one of the paths with some input. We can define the function $reach_G(x)$ as the portion of a graph that is syntactically reachable from the parameter x. The parameter for $reach_G()$ can be a node, an edge, or a set of nodes or edges. Then $reach_G(n_i)$ is the subgraph of G that is syntactically reachable from node n_i, $reach_G(N_0)$ is the subgraph of G that is syntactically reachable from any initial node, $reach_G(e)$ is the subgraph of G syntactically reachable from edge e, and so on. In our use, $reach_G()$ includes the starting nodes. For example, both $reach_G(n_i)$ and $reach_G([n_i, n_j])$ always include n_i, and $reach_G([n_i, n_j])$ includes edge $([n_i, n_j])$. Some graphs have nodes or starting edges that cannot be syntactically reached from any of the initial nodes N_0. These graphs frustrate attempts to satisfy a coverage criterion, so we typically restrict our attention to $reach_G(N_0)$.[1]

Figure 2.3. A single entry single exit graph.

Consider the examples in Figure 2.2. From n_0, it is possible to reach all nodes except n_2 and n_6. From the entire set of initial nodes $\{n_0, n_1, n_2\}$, it is possible to reach all nodes. If we start at n_4, it is possible to reach all nodes except n_0, n_2, n_3, and n_6. If we start at edge (n_6, n_9), it is possible to reach only n_6, n_9 and edge (n_6, n_9). In addition, some graphs (such as finite state machines) have explicit edges from a node to itself, that is, (n_i, n_i).

Basic graph algorithms, usually given in standard data structures texts, can be used to compute syntactic reachability.

A test path represents the execution of a test case. The reason test paths must start in N_0 is that test cases always begin from an initial node. It is important to note that a single test path may correspond to a very large number of test cases on the software. It is also possible that a test path may correspond to zero test cases if the test path is infeasible. We return to the crucial but theoretical issue of infeasibility later, in Section 2.2.1.

Definition 2.31 Test path: A path p, possibly of length zero, that starts at some node in N_0 and ends at some node in N_f.

For some graphs, all test paths start at one node and end at a single node. We call these *single entry/single exit* or *SESE* graphs. For SESE graphs, the set N_0 has exactly one node, called n_0, and the set N_f also has exactly one node, called n_f, which may be the same as n_0. We require that n_f be syntactically reachable from every node in N, and that no node in N (except n_f) be syntactically reachable from n_f (unless n_0 and n_f are the same node). In other words, no edges start at n_f, except when n_0 and n_f happen to be the same node.

Figure 2.3 is an example of a SESE graph. This particular structure is sometimes called a "double-diamond" graph and corresponds to the control flow graph for a sequence of two if-then-else statements. The initial node, n_0, is designated with an incoming arrow (remember we only have one initial node), and the final

Figure 2.4. Test case mappings to test paths.

node, n_6, is designated with a thick circle. Exactly four test paths exist in the double-diamond graph: $[n_0, n_1, n_3, n_4, n_6]$, $[n_0, n_1, n_3, n_5, n_6]$, $[n_0, n_2, n_3, n_4, n_6]$, and $[n_0, n_2, n_3, n_5, n_6]$.

We need some terminology to express the notion of nodes, edges, and subpaths that appear in test paths, and choose familiar terminology from traveling. A test path p is said to *visit* node n if n is in p. Test path p is said to *visit* edge e if e is in p. The term visit applies well to single nodes and edges, but sometimes we want to turn our attention to subpaths. For subpaths, we use the term **tour**. Test path p is said to *tour* subpath q if q is a subpath of p. The first path of Figure 2.3, $[n_0, n_1, n_3, n_4, n_6]$, visits nodes n_0 and n_1, visits edges (n_0, n_1) and (n_3, n_4), and tours the subpath $[n_1, n_3, n_4]$ (among others, these lists are not complete). Since the subpath relationship is reflexive, the tour relationship is also reflexive. That is, any given path p always tours itself.

We define a mapping $path_G$ for tests, so for a test case t, $path_G(t)$ is the test path in graph G that is executed by t. Since it is usually obvious which graph we are discussing, we omit the subscript G. We also define the set of paths toured by a set of tests. For a test set T, *path(T)* is the set of test paths that are executed by the tests in T: $path_G(T) = \{ path_G(t) | t \in T\}$.

Except for nondeterministic structures, which we do not consider until Chapter 7, each test case will tour exactly one test path in graph G. Figure 2.4 illustrates the difference with respect to test case/test path mapping for deterministic vs. nondeterministic software.

Figure 2.5 illustrates a set of test cases and corresponding test paths on a SESE graph with the final node $n_f = n_2$. Some edges are annotated with predicates that describe the conditions under which that edge is traversed. (This notion is formalized later in this chapter.) So, in the example, if a is less than b, the only path is from n_0 to n_1 and then on to n_3 and n_2. This book describes all of the graph coverage criteria in terms of relationships of test paths to the graph in question, but it is important to realize that testing is carried out with test cases, and that the test path is simply a model of the test case in the abstraction captured by the graph.

(a) Graph for testing the case with input integers a, b and output (a+b)

Test case t_1 : (a=0, b=1)	*Map to* →	[Test path p_1 : n_0, n_1, n_3, n_2]
Test case t_2 : (a=1, b=1)	→	[Test path p_2 : n_0, n_3, n_2]
Test case t_3 : (a=2, b=1)	→	[Test path p_3 : n_0, n_2]

(b) Mapping between test cases and test paths

Figure 2.5. A set of test cases and corresponding test paths.

EXERCISES

Section 2.1.

1. Give the sets N, N_0, N_f, and E for the graph in Figure 2.2.
2. Give a path that is not a test path in Figure 2.2.
3. List all test paths in Figure 2.2.
4. In Figure 2.5, find test case inputs such that the corresponding test path visits edge (n_1, n_3).

2.2 GRAPH COVERAGE CRITERIA

The structure in Section 2.1 is adequate to define coverage on graphs. As is usual in the testing literature, we divide these criteria into two types. The first are usually referred to as *control flow coverage* criteria. Because we generalize this situation, we call them *structural graph coverage criteria*. The other criteria are based on the flow of data through the software artifact represented by the graph and are called *data flow coverage* criteria. Following the discussion in Chapter 1, we identify the appropriate test requirements and then define each criterion in terms of the test requirements. In general, for any graph-based coverage criterion, the idea is to identify the test requirements in terms of various structures in the graph.

For graphs, coverage criteria define test requirements, TR, in terms of properties of test paths in a graph G. A typical test requirement is *met* by *visiting* a particular node or edge or by *touring* a particular path. The definitions we have given so far for a *visit* are adequate, but the notion of a *tour* requires more development. We return to the issue of touring later in this chapter and then refine it further in the context

of data flow criteria. The following definition is a refinement of the definition of *coverage* given in Chapter 1:

> *Definition 2.32 Graph Coverage:* Given a set TR of test requirements for a graph criterion C, a test set T satisfies C on graph G if and only if for every test requirement tr in TR, there is at least one test path p in $path(T)$ such that p meets tr.

This is a very general statement that must be refined for individual cases.

2.2.1 Structural Coverage Criteria

We specify graph coverage criteria by specifying a set of test requirements, TR. We will start by defining criteria to visit every node and then every edge in a graph. The first criterion is probably familiar and is based on the old notion of executing every statement in a program. This concept has variously been called "statement coverage," "block coverage," "state coverage," and "node coverage." We use the general graph term "node coverage." Although this concept is familiar and simple, we introduce some additional notation. The notation initially seems to complicate the criterion, but ultimately has the effect of making subsequent criteria cleaner and mathematically precise, avoiding confusion with more complicated situations.

The requirements that are produced by a graph criterion are technically predicates that can have either the value true (the requirement has been met) or false (the requirement has **not** been met). For the double-diamond graph in Figure 2.3, the test requirements for node coverage are: $TR = \{$ *visit* n_0, *visit* n_1, *visit* n_2, *visit* n_3, *visit* n_4, *visit* n_5, *visit* $n_6\}$. That is, we must satisfy a predicate for each node, where the predicate asks whether the node has been visited or not. With this in mind, the formal definition of node coverage is as follows[2]:

> *Definition 2.33 Node Coverage (Formal Definition):* For each node $n \in reach_G(N_0)$, TR contains the predicate "*visit n*."

This notation, although mathematically precise, is too cumbersome for practical use. Thus we choose to introduce a simpler version of the definition that abstracts the issue of predicates in the test requirements.

CRITERION **2.1 Node Coverage (NC):** *TR contains each reachable node in G.*

With this definition, it is left as understood that the term "contains" actually means "contains the predicate $visit_n$." This simplification allows us to simplify the writing of the test requirements for Figure 2.3 to only contain the nodes: $TR = \{n_0, n_1, n_2, n_3, n_4, n_5, n_6\}$. Test path $p_1 = [n_0, n_1, n_3, n_4, n_6]$ meets the first, second, fourth, fifth, and seventh test requirements, and test path $p_2 = [n_0, n_2, n_3, n_5, n_6]$ meets the first, third, fourth, sixth, and seventh. Therefore, if a test set T contains $\{t_1, t_2\}$, where $path(t_1) = p_1$ and $path(t_2) = p_2$, then T satisfies node coverage on G.

The usual definition of node coverage omits the intermediate step of explicitly identifying the test requirements, and is often stated as given below. Notice the economy of the form used above with respect to the standard definition. Several

path (t_1) = [n_0, n_1, n_2]
path (t_2) = [n_0, n_2]

$T_1 = \{ t_1 \}$
T_1 satisfies node coverage on the graph

(a) Node Coverage

$T_2 = \{ t_1 , t_2 \}$
T_2 satisfies edge coverage on the graph

(b) Edge Coverage

Figure 2.6. A graph showing node coverage and edge coverage.

of the exercises emphasize this point by directing the student to recast other criteria in the standard form.

> *Definition 2.34 Node Coverage (NC) (Standard Definition):* Test set T satisfies node coverage on graph G if and only if for every syntactically reachable node n in N, there is some path p in $path(T)$ such that p visits n.

The exercises at the end of the section have the reader reformulate the definitions of some of the remaining coverage criteria in both the formal way and the standard way. We choose the intermediate definition because it is more compact, avoids the extra verbiage in a standard coverage definition, and focuses just on the part of the definition of coverage that changes from criterion to criterion.

Node coverage is implemented in many commercial testing tools, most often in the form of statement coverage. So is the next common criterion of edge coverage, usually implemented as branch coverage:

> CRITERION **2.2 Edge Coverage (EC):** *TR contains each reachable path of length up to 1, inclusive, in G.*

The reader might wonder why the test requirements for edge coverage also explicitly include the test requirements for node coverage – that is, why the phrase "up to" is included in the definition. In fact, all the graph coverage criteria are developed like this. The motivation is subsumption for graphs that do not contain more complex structures. For example, consider a graph with a node that has no edges. Without the "up to" clause in the definition, edge coverage would not cover that node. Intuitively, we would like edge testing to be at least as demanding as node testing. This style of definition is the best way to achieve this property. To make our TR sets readable, we list only the maximal length paths.

Figure 2.6 illustrates the difference between node and edge coverage. In program statement terms, this is a graph of the common "if-else" structure.

Other coverage criteria use only the graph definitions introduced so far. For example, one requirement is that each path of length (up to) two be toured by some test path. With this context, node coverage could be redefined to contain each path of length zero. Clearly, this idea can be extended to paths of any length, although possibly with diminishing returns. We formally define one of these criteria; others are left as exercises for the interested reader.

CRITERION **2.3 Edge-Pair Coverage (EPC):** *TR contains each reachable path of length up to 2, inclusive, in G.*

One useful testing criterion is to start the software in some state (that is, a node in the finite state machine) and then follow transitions (that is, edges) so that the last state is the same as the start state. This type of testing is used to verify that the system is not changed by certain inputs. Shortly we will formalize this notion as round trip coverage.

Before defining round trip coverage, we need a few more definitions. A path from n_i to n_j is *simple* if no node appears more than once in the path, with the exception that the first and last nodes may be identical. That is, simple paths have no internal loops, although the entire path itself may wind up being a loop. One useful aspect of simple paths is that any path can be created by composing simple paths.

Even fairly small programs may have a very large number of simple paths. Most of these simple paths aren't worth addressing explicitly since they are subpaths of other simple paths. For a coverage criterion for simple paths we would like to avoid enumerating the entire set of simple paths. To this end we list only maximal length simple paths. To clarify this notion, we introduce a formal definition for a maximal length simple path, which we call a *prime path*, and we adopt the name "prime" for the criterion:

Definition 2.35 Prime Path: A path from n_i to n_j is a prime path if it is a simple path and it does not appear as a proper subpath of any other simple path.

CRITERION **2.4 Prime Path Coverage (PPC):** *TR contains each prime path in G.*

While this definition of prime path coverage has the practical advantage of keeping the number of test requirements down, it suffers from the problem that a given infeasible prime path may well incorporate many feasible simple paths. The solution is direct: replace the infeasible prime path with relevant feasible subpaths. For the purposes of this textbook, we choose not to include this aspect of prime path coverage formally in the definition, but we assume it in later theoretical characterizations of prime path coverage.

Prime path coverage has two special cases that we include below for historical reasons. From a practical perspective, it is usually better simply to adopt prime path coverage. Both special cases involve treatment of loops with "round trips."

A *round trip* path is a prime path of nonzero length that starts and ends at the same node. One type of round trip test coverage requires at least one round trip path to be taken for each node, and another requires all possible round trip paths.

CRITERION **2.5 Simple Round Trip Coverage (SRTC):** *TR contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path.*

CRITERION **2.6 Complete Round Trip Coverage (CRTC):** *TR contains all round-trip paths for each reachable node in G.*

Next we turn to path coverage, which is traditional in the testing literature.

CRITERION **2.7 Complete Path Coverage (CPC):** *TR contains all paths in G.*

Sadly, complete path coverage is useless if a graph has a cycle, since this results in an infinite number of paths, and hence an infinite number of test requirements. A variant of this criterion is, however, useful. Suppose that instead of requiring all paths, we consider a specified set of paths. For example, these paths might be given by a customer in the form of usage scenarios.

CRITERION **2.8 Specified Path Coverage (SPC):** *TR contains a set S of test paths, where S is supplied as a parameter.*

Complete path coverage is not feasible for graphs with cycles; hence the reason for developing the other alternatives listed above. Figure 2.7 contrasts prime path coverage with complete path coverage. Part (a) of the figure shows the "diamond" graph, which contains no loops. Both complete path coverage and prime path coverage can be satisfied on this graph with the two paths shown. Part (b), however, includes a loop from n_1 to n_3 to n_4 to n_1, thus the graph has an infinite number of possible test paths, and complete path coverage is not possible. The requirements for prime path coverage, however, can be toured with two test paths, for example, $[n_0, n_1, n_2]$ and $[n_0, n_1, n_3, n_4, n_1, n_3, n_4, n_1, n_2]$.

Touring, Sidetrips, and Detours

An important but subtle point to note is that while simple paths do not have internal loops, we do **not** require the test paths that tour a simple path to have this property. That is, we distinguish between the path that **specifies** a test requirement and the portion of the test path that **meets** the requirement. The advantage of separating these two notions has to do with the issue of infeasible test requirements. Before describing this advantage, let us refine the notion of a tour.

We previously defined "visits" and "tours," and recall that using a path p to tour a subpath $[n_1, n_2, n_3]$ means that the subpath is a subpath of p. This is a rather strict definition because each node and edge in the subpath must be visited **exactly** in the order that they appear in the subpath. We would like to relax this a bit to allow

Prime Paths = { [n_0, n_1, n_3], [n_0, n_2, n_3] }
path (t_1) = [n_0, n_1, n_3]
path (t_2) = [n_0, n_2, n_3]
T_1 = {t_1, t_2}
T_1 satisfies prime path coverage on the graph

(a) Prime Path Coverage on a Graph with No Loops

Prime Paths = { [n_0, n_1, n_2],
[n_0, n_1, n_3, n_4], [n_1, n_3, n_4, n_1],
[n_3, n_4, n_1, n_3], [n_4, n_1, n_3, n_4],
[n_3, n_4, n_1, n_2] }
path (t_3) = [n_0, n_1, n_2]
path (t_4) = [n_0, n_1, n_3, n_4, n_1, n_3, n_4, n_1, n_2]
T_2 = {t_3, t_4}
T_2 satisfies prime path coverage on the graph

(b) Prime Path Coverage on a Graph with Loops

Figure 2.7. Two graphs showing prime path coverage.

loops to be included in the tour. Consider the graph in Figure 2.8, which features a small loop from b to c and back.

If we are required to tour subpath $q = [a, b, d]$, the strict definition of tour prohibits us from meeting the requirement with any path that contains c, such as $p = [s_0, a, b, c, b, d, s_f]$, because we do not visit a, b, and d in exactly the same order. We relax the tour definition in two ways. The first allows the tour to include "sidetrips," where we can leave the path temporarily from a node and then return to the same node. The second allows the tour to include more general "detours" where we can leave the path from a node and then return to the **next** node on the

Figure 2.8. Graph with a loop.

(a) Graph being toured with a sidetrip

(b) Graph being toured with a detour

Figure 2.9. Tours, sidetrips, and detours in graph coverage.

path (skipping an edge). In the following definitions, q is a required subpath that is assumed to be simple.

Definition 2.36 Tour: Test path p is said to *tour* subpath q if and only if q is a subpath of p.

Definition 2.37 Tour with Sidetrips: Test path p is said to *tour* subpath q *with sidetrips* if and only if every **edge** in q is also in p in the same order.

Definition 2.38 Tour with Detours: Test path p is said to *tour* subpath q *with detours* if and only if every **node** in q is also in p in the same order.

The graphs in Figure 2.9 illustrate sidetrips and detours on the graph from Figure 2.8. In Figure 2.9(a), the dashed lines show the sequence of edges that are executed in a tour with a sidetrip. The numbers on the dashed lines indicate the order in which the edges are executed. In Figure 2.9(b), the dashed lines show the sequence of edges that are executed in a tour with a detour.

While these differences are rather small, they have far-reaching consequences. The difference between sidetrips and detours can be seen in Figure 2.9. The subpath $[b, c, b]$ is a **sidetrip** to $[a, b, d]$ because it leaves the subpath at node b and then returns to the subpath at node b. Thus, every edge in the subpath $[a, b, d]$ is executed in the same order. The subpath $[b, c, d]$ is a **detour** to $[a, b, d]$ because it leaves the subpath at node b and then returns to a node in the subpath at a later point, bypassing the edge (b, d). That is, every node $[a, b, d]$ is executed in the same order but every edge is not. Detours have the potential to drastically change the behavior of the intended test. That is, a test that takes the edge (c, d) may exhibit different

behavior and test different aspects of the program than a test that takes the edge (b, d).

To use the notion of sidetrips and detours, one can "decorate" each appropriate graph coverage criterion with a choice of touring. For example, prime path coverage could be defined strictly in terms of tours, less strictly to allow sidetrips, or even less strictly to allow detours.

The position taken in this book is that sidetrips are a practical way to deal with infeasible test requirements, as described below. Hence we include them explicitly in our criteria. Detours seem less practical, and so we do not include them further.

Dealing with Infeasible Test Requirements

If sidetrips are not allowed, a large number of infeasible requirements can exist. Consider again the graph in Figure 2.9. In many programs it will be impossible to take the path from a to d without going through node c at least once because, for example, the loop body is written such that it cannot be skipped. If this happens, we need to allow sidetrips. That is, it may not be possible to tour the path $[a, b, d]$ without a sidetrip.

The argument above suggests dropping the strict notion of touring and simply allowing test requirements to be met with sidetrips. However, this is not always a good idea! Specifically, if a test requirement can be met without a sidetrip, then doing so is clearly superior to meeting the requirement with a sidetrip. Consider the loop example again. If the loop can be executed zero times, then the path $[a, b, d]$ should be toured without a sidetrip.

The argument above suggests a hybrid treatment with desirable practical and theoretical properties. The idea is to meet test requirements first with strict tours, and then allow sidetrips for unmet test requirements. Clearly, the argument could easily be extended to detours, but, as mentioned above, we elect not to do so.

> *Definition 2.39 Best Effort Touring:* Let TR_{tour} be the subset of test requirements that can be toured and $TR_{sidetrip}$ be the subset of test requirements that can be toured with sidetrips. Note that $TR_{tour} \subseteq TR_{sidetrip}$. A set T of test paths achieves *best effort touring* if for every path p in TR_{tour}, some path in T tours p directly and for every path p in $TR_{sidetrip}$, some path in T tours p either directly or with a sidetrip.

Best-effort touring has the practical benefit that as many test requirements are met as possible, yet each test requirement is met in the strictest possible way. As we will see in Section 2.2.3 on subsumption, best-effort touring has desirable theoretical properties with respect to subsumption.

Finding Prime Test Paths

It turns out to be relatively simple to find all prime paths in a graph, and test paths to tour the prime paths can be constructed in a mechanical manner. Consider the example graph in Figure 2.10. It has seven nodes and nine edges, including a loop and an edge from node n_4 to itself (sometimes called a "self-loop.")

Prime paths can be found by starting with paths of length 0, then extending to length 1, and so on. Such an algorithm collects all simple paths, whether prime or

Figure 2.10. An example for prime test paths.

not. The prime paths can be easily screened from this set. The set of paths of length 0 is simply the set of nodes, and the set of paths of length 1 is simply the set of edges. For simplicity, we simply list the node numbers in this example.

Simple paths of length 0 (7):

1) [0]
2) [1]
3) [2]
4) [3]
5) [4]
6) [5]
7) [6] !

The exclamation point on the path [6] tells us that this path cannot be extended. Specifically, the final node 6 has no outgoing edges, and so paths that end with 6 are not extended further.

Simple paths of length 1 (9):

8) [0, 1]
9) [0, 4]
10) [1, 2]
11) [1, 5]
12) [2, 3]
13) [3, 1]
14) [4, 4] *
15) [4, 6] !
16) [5, 6] !

The asterisk on the path [4, 4] tells us that path can go no further because the first node is the same as the last (it is already a cycle). For paths of length 2, we identify each path of length 1 that is not a cycle (marked with asterisks). We then extend the path with every node that can be reached from the final node in the path unless that node is already in the path and not the first node. The first path of length 1, [0, 1], is extended to [0, 1, 2] and [0, 1, 5]. The second, [0, 4], is extended to [0, 4, 6] but not [0, 4, 4], because node 4 is already in the path ([0, 4, 4] is not simple and thus is not prime).

Simple paths of length 2 (8):

17) [0, 1, 2]
18) [0, 1, 5]
19) [0, 4, 6] !
20) [1, 2, 3]
21) [1, 5, 6] !
22) [2, 3, 1]
23) [3, 1, 2]
24) [3, 1, 5]

Paths of length 3 are computed in a similar way.
Simple paths of length 3 (7):

25) [0, 1, 2, 3] !
26) [0, 1, 5, 6] !
27) [1, 2, 3, 1] *
28) [2, 3, 1, 2] *
29) [2, 3, 1, 5]
30) [3, 1, 2, 3] *
31) [3, 1, 5, 6] !

Finally, only one path of length 4 exists. Three paths of length 3 cannot be extended because they are cycles; two others end with node 6. Of the remaining two, the path that ends in node 3 cannot be extended because [0, 1, 2, 3, 1] is **not** simple and thus is not prime.
Prime paths of length 4 (1):

32) [2, 3, 1, 5, 6]!

The prime paths can be computed by eliminating any path that is a (proper) subpath of some other simple path. Note that every simple path without an exclamation mark or asterisk is eliminated as it can be extended and is thus a proper subpath of some other simple path. There are eight prime paths:

14) [4, 4] *
19) [0, 4, 6] !
25) [0, 1, 2, 3] !
26) [0, 1, 5, 6] !
27) [1, 2, 3, 1] *
28) [2, 3, 1, 2] *
30) [3, 1, 2, 3] *
32) [2, 3, 1, 5, 6]!

This process is guaranteed to terminate because the length of the longest possible prime path is the number of nodes. Although graphs often have many simple paths (32 in this example, of which 8 are prime), they can usually be toured with far fewer test paths. Many possible algorithms can find test paths to tour the prime paths. Observation will suffice with a graph as simple as in Figure 2.10. For example, it can be seen that the four test paths [0, 1, 5, 6], [0, 1, 2, 3, 1, 2, 3, 1, 5, 6], [0, 4, 6],

and [0, 4, 4, 6] are enough. This approach, however, is error-prone. The easiest thing to do is to tour the loop [1, 2, 3] only once, which omits the prime paths [2, 3, 1, 2] and [3, 1, 2, 3].

With more complicated graphs, a mechanical approach is needed. We recommend starting with the longest prime paths and extending them to the beginning and end nodes in the graph. For our example, this results in the test path [0, 1, 2, 3, 1, 5, 6]. The test path [0, 1, 2, 3, 1, 5, 6] tours 3 prime paths 25, 27, and 32.

The next test path is constructed by extending one of the longest remaining prime paths; we will continue to work backward and choose 30. The resulting test path is [0, 1, 2, 3, 1, 2, 3, 1, 5, 6], which tours 2 prime paths, 28 and 30 (it also tours paths 25 and 27).

The next test path is constructed by using the prime path 26 [0, 1, 5, 6]. This test path tours only maximal prime path 26.

Continuing in this fashion yields two more test paths, [0, 4, 6] for prime path 19, and [0, 4, 4, 6] for prime path 14.

The complete set of test paths is then:

1) [0, 1, 2, 3, 1, 5, 6]
2) [0, 1, 2, 3, 1, 2, 3, 1, 5, 6]
3) [0, 1, 5, 6]
4) [0, 4, 6]
5) [0, 4, 4, 6]

This can be used as is, or optimized if the tester desires a smaller test set. It is clear that test path 2 tours the prime paths toured by test path 1, so 1 can be eliminated, leaving the four test paths identified informally earlier in this section. Simple algorithms can automate this process.

EXERCISES

Section 2.2.1.

1. Redefine *edge coverage* in the standard way (see the discussion for *node coverage*).
2. Redefine *complete path coverage* in the standard way (see the discussion for *node coverage*).
3. Subsumption has a significant weakness. Suppose criterion C_{strong} subsumes criterion C_{weak} and that test set T_{strong} satisfies C_{strong} and test set T_{weak} satisfies C_{weak}. It is not necessarily the case that T_{weak} is a subset of T_{strong}. It is also not necessarily the case that T_{strong} reveals a fault if T_{weak} reveals a fault. Explain these facts.
4. Answer questions (a)–(d) for the graph defined by the following sets:
   - $N = \{1, 2, 3, 4\}$
   - $N_0 = \{1\}$
   - $N_f = \{4\}$
   - $E = \{(1, 2), (2, 3), (3, 2), (2, 4)\}$

(a) Draw the graph.
(b) List test paths that achieve node coverage, but not edge coverage.
(c) List test paths that achieve edge coverage, but not edge Pair coverage.
(d) List test paths that achieve edge pair coverage.

5. Answer questions (a)–(g) for the graph defined by the following sets:
   - $N = \{1, 2, 3, 4, 5, 6, 7\}$
   - $N_0 = \{1\}$
   - $N_f = \{7\}$
   - $E = \{(1, 2), (1, 7), (2, 3), (2, 4), (3, 2), (4, 5), (4, 6), (5, 6), (6, 1)\}$

   Also consider the following (candidate) test paths:
   - $t_0 = [1, 2, 4, 5, 6, 1, 7]$
   - $t_1 = [1, 2, 3, 2, 4, 6, 1, 7]$

   (a) Draw the graph.
   (b) List the test requirements for edge-pair coverage. (Hint: You should get 12 requirements of length 2).
   (c) Does the given set of test paths satisfy edge-pair coverage? If not, identify what is missing.
   (d) Consider the simple path **[3, 2, 4, 5, 6]** and test path **[1, 2, 3, 2, 4, 6, 1, 2, 4, 5, 6, 1, 7]**. Does the test path tour the simple path directly? With a sidetrip? If so, identify the sidetrip.
   (e) List the test requirements for node coverage, edge coverage, and prime path coverage on the graph.
   (f) List test paths that achieve node coverage but not edge coverage on the graph.
   (g) List test paths that achieve edge coverage but not prime path coverage on the graph.

6. Answer questions (a)–(c) for the graph in Figure 2.2.
   (a) Enumerate the test requirements for node coverage, edge coverage, and prime path coverage on the graph.
   (b) List test paths that achieve node coverage but not edge coverage on the graph.
   (c) List test paths that achieve edge coverage but not prime path coverage on the graph.

7. Answer questions (a)–(d) for the graph defined by the following sets:
   - $N = \{0, 1, 2\}$
   - $N_0 = \{0\}$
   - $N_f = \{2\}$
   - $E = \{(0, 1), (0, 2), (1, 0), (1, 2), (2, 0)\}$

   Also consider the following (candidate) paths:
   - $p_0 = [0, 1, 2, 0]$
   - $p_1 = [0, 2, 0, 1, 2]$
   - $p_2 = [0, 1, 2, 0, 1, 0, 2]$
   - $p_3 = [1, 2, 0, 2]$
   - $p_4 = [0, 1, 2, 1, 2]$

   (a) Which of the listed paths are test paths? Explain the problem with any path that is not a test path.

Figure 2.11. A graph showing variables, def sets and use sets.

(b) List the eight test requirements for edge-pair coverage (only the length two subpaths).
(c) Does the set of **test** paths (part a) above satisfy edge-pair coverage? If not, identify what is missing.
(d) Consider the prime path $[n_2, n_0, n_2]$ and path p_2. Does p_2 tour the prime path directly? With a sidetrip?

8. Design and implement a program that will compute all prime paths in a graph, then derive test paths to tour the prime paths. Although the user interface can be arbitrarily complicated, the simplest version will be to accept a graph as input by reading a list of nodes, initial nodes, final nodes, and edges.

2.2.2 Data Flow Criteria

The next few testing criteria are based on the assumption that to test a program adequately, we should focus on the flows of data values. Specifically, we should try to ensure that the values created at one point in the program are created and used correctly. This is done by focusing on definitions and uses of values. A *definition (def)* is a location where a value for a variable is stored into memory (assignment, input, etc.). A *use* is a location where a variable's value is accessed. Data flow testing criteria use the fact that values are carried from defs to uses. We call these *du-pairs* (they are also known as *definition-use*, *def-use*, and *du* associations in the testing literature). The idea of data flow criteria is to exercise du-pairs in various ways.

First we must integrate data flow into the existing graph model. Let V be a set of variables that are associated with the program artifact being modeled in the graph. Each node n and edge e is considered to define a subset of V; this set is called $def(n)$ or $def(e)$. (Although graphs from programs cannot have defs on edges, other software artifacts such as finite state machines can allow defs as side effects on edges.) Each node n and edge e is also considered to use a subset of V; this set is called $use(n)$ or $use(e)$.

Figure 2.11 gives an example of a graph annotated with defs and uses. All variables involved in a decision are assumed to be used on the associated edges, so a and b are in the use set of all three edges (n_0, n_1), (n_0, n_2), and (n_0, n_3).

An important concept when discussing data flow criteria is that a def of a variable may or may not reach a particular use. The most obvious reason that a def of a variable v at location l_i (a location could be a node or an edge) will not reach a use at location l_j is because no path goes from l_i to l_j. A more subtle reason is that the variable's value may be changed by another def before it reaches the use. Thus, a path from l_i to l_j is *def-clear* with respect to variable v if for every node n_k and every edge e_k on the path, $k \neq i$ and $k \neq j$, v is not in $def(n_k)$ or in $def(e_k)$. That is, no location between l_i and l_j changes the value. If a def-clear path goes from l_i to l_j with respect to v, we say that the def of v at l_i *reaches* the use at l_j.

For simplicity, we will refer to the start and end of a du-path as nodes, even if the definition or the use occurs on an edge. We discuss relaxing this convention later. Formally, a *du-path* with respect to a variable v is a simple path that is def-clear with respect to v from a node n_i for which v is in $def(n_i)$ to a node n_j for which v is in $use(n_j)$. We want the paths to be simple to ensure a reasonably small number of paths. Note that a du-path is always associated with a specific variable v, a du-path always has to be simple, and there may be intervening uses on the path.

Figure 2.12 gives an example of a graph annotated with defs and uses. Rather than displaying the actual sets, we show the full program statements that are associated with the nodes and edges. This is common and often more informative to a human, but the actual sets are simpler for automated tools to process. Note that the parameters (*subject* and *pattern*) are considered to be *explicitly defined* by the first node in the graph. That is, the def set of node 1 is $def(1) = \{subject, pattern\}$. Also note that decisions in the program (for example, *if subject*[*iSub*] == *pattern*[0]) result in uses of each of the associated variables for both edges in the decision. That is, $use(4, 10) \equiv use(4,5) \equiv \{subject, iSub, pattern\}$. The parameter *subject* is used at node 2 (with a reference to its *length* attribute) and at edges (4, 5), (4, 10), (7, 8), and (7, 9), thus du-paths exist from node 1 to node 2 and from node 1 to each of those four edges.

Figure 2.13 shows the same graph, but this time with the def and use sets explicitly marked on the graph.[3] Note that node 9 both defines and uses the variable *iPat*. This is because of the statement *iPat* ++, which is equivalent to *iPat* = *iPat*+1. In this case, the use occurs before the def, so for example, a def-clear path goes from node 5 to node 9 with respect to *iPat*.

The test criteria for data flow will be defined as sets of du-paths. This makes the criteria quite simple, but first we need to categorize the du-paths into several *groups*.

The first grouping of du-paths is according to definitions. Specifically, consider all of the du-paths with respect to a given variable defined in a given node. Let the *def-path* set $du(n_i, v)$ be the set of du-paths with respect to variable v that start at node n_i. Once we have clarified the notion of touring for dataflow coverage, we will define the All-Defs criterion by simply asking that at least one du-path from each def-path set be toured. Because of the large number of nodes in a typical graph, and the potentially large number of variables defined at each node, the number of def-path sets can be quite large. Even so, the coverage criterion that arises from the def-path groupings tends to be quite weak.

Perhaps surprisingly, it is *not* helpful to group du-paths by uses, and so we will not provide a definition of "use-path" sets that parallels the definition of def-path sets given above.

Figure 2.12. A graph showing an example of du-paths.

The second, and more important, grouping of du-paths is according to pairs of definitions and uses. We call this the *def-pair* set. After all, the heart of data flow testing is allowing definitions to flow to uses. Specifically, consider all of the du-paths with respect to a given variable that are defined in one node and used in another (possibly identical) node. Formally, let the *def-pair* set $du(n_i, n_j, v)$ be the set of du-paths with respect to variable v that start at node n_i and end at node n_j. Informally, a def-pair set collects together all the (simple) ways to get from a given definition to a given use. Once we have clarified the notion of touring for dataflow coverage, we will define the All-Uses criterion by simply asking that at least one du-path from

Figure 2.13. Graph showing explicit def and use sets.

each def-pair set be toured. Since each definition can typically reach multiple uses, there are usually many more def-pair sets than def-path sets.

In fact, the def-pair set for a def at node n_i is the union of all the def-path sets for that def. More formally: $du(n_i, v) = \cup_{n_j} du(n_i, n_j, v)$.

To illustrate the notions of def-path sets and def-pair sets, consider du-paths with respect to the variable *iSub*, which has one of its definitions in node 10 in Figure 2.13. There are du-paths with respect to *iSub* from node 10 to nodes 5 and 10, and to edges (3, 4), (3, 11), (4, 5), (4, 10), (7, 8), and (7, 9).

The def-path set for the use of *isub* at node 10 is:

$$du(10, iSub) = \{[10, 3, 4], [10, 3, 4, 5], [10, 3, 4, 5, 6, 7, 8], [10, 3, 4, 5, 6, 7, 9], [10, 3, 4, 5, 6, 10], [10, 3, 4, 5, 6, 7, 8, 10], [10, 3, 4, 10], [10, 3, 11]\}$$

This def-path set can be broken up into the following def-pair sets:

$$\begin{aligned} du(10, 4, iSub) &= is\{[10, 3, 4]\} \\ du(10, 5, iSub) &= \{[10, 3, 4, 5]\} \\ du(10, 8, iSub) &= \{[10, 3, 4, 5, 6, 7, 8]\} \\ du(10, 9, iSub) &= \{[10, 3, 4, 5, 6, 7, 9]\} \\ du(10, 10, iSub) &= \{[10, 3, 4, 5, 6, 10], [10, 3, 4, 5, 6, 7, 8, 10], [10, 3, 4, 10]\} \\ du(10, 11, iSub) &= \{[10, 3, 11]\} \end{aligned}$$

Next, we extend the definition of *tour* to apply to du-paths. A test path p is said to *du tour* subpath d with respect to v if p tours d and the portion of p to which d corresponds is def-clear with respect to v. Depending on how one wishes to define the coverage criteria, one can either allow or disallow def-clear sidetrips with respect to v when touring a du-path. Because def-clear sidetrips make it possible to tour more du-paths, we define the dataflow coverage criteria given below to allow sidetrips where necessary.

Now we can define the primary data flow coverage criteria. The three most common are best understood informally. The first requires that each def reaches **at least one use**, the second requires that each def reaches **all possible uses**, and the third requires that each def reaches all possible uses through **all possible du-paths**. As mentioned in the development of def-path sets and def-pair sets, the formal definitions of the criteria are simply appropriate selections from the appropriate set. For each test criterion below, we assume best effort touring (see Section 2.2.1), where sidetrips are required to be def-clear with respect to the variable in question.

> CRITERION **2.9 All-Defs Coverage (ADC):** *For each def-path set $S = du(n, v)$, TR contains at least one path d in S.*

Remember that the def-path set $du(n, v)$ represents all def-clear simple paths from n to all uses of v. So All-Defs requires us to tour at least one path to at least one use.

> CRITERION **2.10 All-Uses Coverage (AUC):** *For each def-pair set $S = du(n_i, n_j, v)$, TR contains at least one path d in S.*

Remember that the def-pair set $du(n_i, n_j, v)$ represents all the def-clear simple paths from a def of v at n_i to a use of v at n_j. So All-Uses requires us to tour at least one path for every def-use pair.[4]

> CRITERION **2.11 All-du-Paths Coverage (ADUPC):** *For each def-pair set $S = du(n_i, n_j, v)$, TR contains every path d in S.*

The definition could also simply be written as "include every du-path." We chose the given formulation because it highlights that the key difference between All-Uses

Figure 2.14. Example of the differences among the three data flow coverage criteria.

and All-du-Paths is a change in quantifier. Specifically, the "at least one du-path" directive in All-Uses is changed to "every path" in All-du-Paths. Thought of in terms of def-use pairs, All-Uses requires *some* def-clear simple path to each use, whereas All-du-Paths requires *all* def-clear simple paths to each use.

To simplify the development above, we assumed that definitions and uses occurred on nodes. Naturally, definitions and uses can occur on edges as well. It turns out that the development above also works for uses on edges, so data flow on program flow graphs can be easily defined (uses on program flow graph edges are sometimes called "p-uses"). However, the development above does not work if the graph has definitions on edges. The problem is that a du-path from an edge to an edge is no longer necessarily simple, since instead of simply having a common first and last *node*, such a du-path now might have a common first and last *edge*. It is possible to modify the definitions to explicitly mention definitions and uses on edges as well as nodes, but the definitions tend to get messier. The bibliographic notes contain pointers for this type of development.

Figure 2.14 illustrates the differences among the three data flow coverage criteria with the double-diamond graph. The graph has one def, so only one path is needed to satisfy all-defs. The def has two uses, so two paths are needed to satisfy all-uses. Since two paths go from the def to each use, four paths are needed to satisfy all-du-paths. Note that the definitions of the data flow criteria leave open the choice of touring. The literature uses various choices – in some cases demanding direct touring, and, in other cases, allowing def-clear sidetrips. Our recommendation is best-effort touring, a choice that, in contrast to the treatments in the literature, yields the desired subsumption relationships even in the case of infeasible test

requirements. From a practical perspective, best-effort touring also makes sense – each test requirement is satisfied as rigorously as possible.

2.2.3 Subsumption Relationships among Graph Coverage Criteria

Recall from Chapter 1 that coverage criteria are often related to one another by *subsumption*. The first relation to note is that edge coverage subsumes node coverage. In most cases, this is because if we traverse every edge in a graph, we will visit every node. However, if a graph has a node with no incoming or outgoing edges, traversing every edge will not reach that node. Thus, edge coverage is defined to include every path of length **up to 1**, that is, of length 0 (all nodes) and length 1 (all edges). The subsumption does not hold in the reverse direction. Recall that Figure 2.6 gave an example test set that satisfied node coverage but not edge coverage. Hence, node coverage does not subsume edge coverage.

We have a variety of subsumption relations among the criteria. Where applicable, the structural coverage relations assume best-effort touring. Because best-effort Touring is assumed, the subsumption results hold even if some test requirements are infeasible.

The subsumption results for data flow criteria are based on three assumptions: (1) every use is preceded by a def, (2) every def reaches at least one use, and (3) for every node with multiple outgoing edges, at least one variable is used on each out edge, and the same variables are used on each out edge. If we satisfy All-Uses

Figure 2.15. Subsumption relations among graph coverage criteria.

coverage, then we will have implicitly ensured that every def was used. Thus All-Defs is also satisfied and All-Uses subsumes All-Defs. Likewise, if we satisfy All-du-Paths coverage, then we will have implicitly ensured that every def reached every possible use. Thus All-Uses is also satisfied and All-du-Paths subsumes All-Uses. Additionally, each edge is based on the satisfaction of some predicate, so each edge has at least one use. Therefore All-Uses will guarantee that each edge is executed at least once, so All-Uses subsumes edge coverage.

Finally, each du-path is also a simple path, so prime path coverage subsumes All-du-Paths coverage.[5] This is a significant observation, since computing prime paths is considerably simpler than analyzing data flow relationships. Figure 2.15 shows the subsumption relationships among the structural and data flow coverage criteria.

EXERCISES

Section 2.2.3.

1. Below are four graphs, each of which is defined by the sets of nodes, initial nodes, final nodes, edges, and defs and uses. Each graph also contains a collection of test paths. Answer the following questions about each graph.

Graph I.
$N = \{0, 1, 2, 3, 4, 5, 6, 7\}$
$N_0 = \{0\}$
$N_f = \{7\}$
$E = \{(0,1), (1,2), (1,7), (2,3), (2,4), (3,2), (4,5), (4,6), (5,6), (6,1)\}$
$def(0) = def(3) = use(5) = use(7) = \{x\}$

Test Paths:
$t1 = [0, 1, 7]$
$t2 = [0, 1, 2, 4, 6, 1, 7]$
$t3 = [0, 1, 2, 4, 5, 6, 1, 7]$
$t4 = [0, 1, 2, 3, 2, 4, 6, 1, 7]$
$t5 = [0, 1, 2, 3, 2, 3, 2, 4, 5, 6, 1, 7]$
$t6 = [0, 1, 2, 3, 2, 4, 6, 1, 2, 4, 5, 6, 1, 7]$

Graph II.
$N = \{1, 2, 3, 4, 5, 6\}$
$N_0 = \{1\}$
$N_f = \{6\}$
$E = \{(1,2), (2,3), (2,6), (3,4), (3,5), (4,5), (5,2)\}$
$def(x) = \{1, 3\}$
$use(x) = \{3, 6\}$ // Assume the use of x in 3 precedes the def
Test Paths:
$t1 = [1, 2, 6]$
$t2 = [1, 2, 3, 4, 5, 2, 3, 5, 2, 6]$
$t3 = [1, 2, 3, 5, 2, 3, 4, 5, 2, 6]$
$t4 = [1, 2, 3, 5, 2, 6]$

Graph III.
$N = \{1, 2, 3, 4, 5, 6\}$
$N_0 = \{1\}$
$N_f = \{6\}$
$E = \{(1,2), (2,3), (3,4), (3,5), (4,5), (5,2), (2,6)\}$
$def(x) = \{1, 4\}$
$use(x) = \{3, 5, 6\}$

Test Paths:
$t_1 = [1, 2, 3, 5, 2, 6]$
$t_2 = [1, 2, 3, 4, 5, 2, 6]$

Graph IV.
$N = \{1, 2, 3, 4, 5, 6\}$
$N_0 = \{1\}$
$N_f = \{6\}$
$E = \{(1,2), (2,3), (2,6), (3,4), (3,5), (4,5), (5,2)\}$
$def(x) = \{1, 5\}$
$use(x) = \{5, 6\}$ // Assume the use of x in 5 precedes the def
Test Paths:
$t1 = [1, 2, 6]$
$t2 = [1, 2, 3, 4, 5, 2, 3, 5, 2, 6]$
$t3 = [1, 2, 3, 5, 2, 3, 4, 5, 2, 6]$

(a) Draw the graph.
(b) List all of the du-paths with respect to x. (Note: Include all du-paths, even those that are subpaths of some other du-path).
(c) For each test path, determine which du-paths that test path tours. For this part of the exercise, you should consider both direct touring and sidetrips. Hint: A table is a convenient format for describing this relationship.
(d) List a minimal test set that satisfies *all-defs* coverage with respect to x. (Direct tours only.) Use the given test paths.
(e) List a minimal test set that satisfies *all-uses* coverage with respect to x.

Figure 2.16. CFG fragment for the if-else structure.

(Direct tours only.) Use the given test paths.

(f) List a minimal test set that satisfies *all-du-paths* coverage with respect to x. (Direct tours only.) Use the given test paths.

2.3 GRAPH COVERAGE FOR SOURCE CODE

Most of the graph coverage criteria were developed for source code, and these definitions match the definitions in Section 2.2 very closely. As in Section 2.2, we first consider structural coverage criteria and then data flow criteria.

2.3.1 Structural Graph Coverage for Source Code

The most widely used graph coverage criteria are defined on source code. Although precise details vary from one programming language to another, the basic pattern is the same for most common languages. To apply one of the graph criteria, the first step is to define the graph, and for source code, the most common graph is called a *control flow graph (CFG)*. Control flow graphs associate an edge with each possible branch in the program, and a node with sequences of statements. Formally, a *basic block* is a maximum sequence of program statements such that if any one statement of the block is executed, all statements in the block are executed. A basic block has only one entry point and one exit point. Our first example language structure is an if statement with an else clause, shown as Java code followed by the corresponding CFG in Figure 2.16. The if-else structure results in two basic blocks.

Note that the two statements in the then part of the if statement both appear in the same node. Node n_0, which represents the conditional test `x < y` has more than one out-edge, and is called a *decision* node. Node n_3, which has more than one in-edge, is called a *junction* node.

Next we turn to the degenerate case of an if statement without an else clause, shown in Figure 2.17. This is the same graph previously seen in Figure 2.6, but this time based on actual program statements.

Note that the control flow graph for this structure has only three nodes. The reader should note that a test with $x < y$ traverses all of the nodes in this control flow graph, but not all of the edges.

Representing loops is a little tricky because we have to include nodes that are not directly derived from program statements. The simplest kind of loop is a while loop with an initializing statement, as shown in Figure 2.18. (Assume that y has a

Figure 2.17. CFG fragment for the if structure without an else.

value defined at this point in the program.)

The graph for the while structure has a decision node, which is needed for the conditional test, and a single node for the body of the while loop. Node n_1 is sometimes called a "dummy node," because it does not represent any statements, but gives the iteration edge (n_2, n_1) somewhere to go. Node n_1 can also be thought of as representing a decision. A common mistake for beginners is to try to have the edge go to n_0; this is not correct because that would mean the initialization step is done each iteration of the loop. Note that the method call f(x,y) is not expanded in this particular graph; we return to this issue later.

Now, consider a for loop that is equivalent to the prior while loop. The graph becomes a little more complicated, as shown in Figure 2.19, essentially because the for structure is at a very high level of abstraction.

Although the initialization, test, and increment of the loop control variable x are all on the same line in the program, they need to be associated with different nodes in the graph. The control flow graph for the for loop is slightly different from that of the while loop. Specifically, we show the increment of x in a different node than the method call y = f(x,y). Technically speaking, this violates the definition of a basic block and the two nodes should be combined, but it is often easier to develop templates for the various possible program structures and then plug the control flow graph for the relevant code into the correct spot in the template. Commercial tools typically do this to make the graph generation simpler. In fact, commercial tools often do not follow the strict definition of the basic block and sometimes add seemingly random nodes. This can have trivial effects on the bookkeeping (for ex-

Figure 2.18. CFG fragment for the while loop structure.

Figure 2.19. CFG fragment for the for loop structure.

ample, we might cover 67 of 73 instead of 68 of 75), but is not really important for testing.

Our final language structure is the case statement, or switch in Java. The case structure can be graphed either as a single node with multi-way branching or as a series of if-then-else structures. We choose to illustrate the case structure with multi-way branching, as in Figure 2.20.

The coverage criteria from the previous section can now be applied to graphs from source code. The application is direct with only the names being changed. Node coverage is often called *statement coverage* or *basic block coverage*, and edge coverage is often called *branch coverage*.

2.3.2 Data Flow Graph Coverage for Source Code

This section applies the data flow criteria to the code examples given in the prior section. Before we can do this, we need to define what constitutes a *def* and what constitutes a *use*. A *def* is a location in the program where a value for a variable is stored into memory (assignment, input, etc.). A *use* is a location where a variable's

Figure 2.20. CFG fragment for the case structure.

value is accessed.

A def may occur for variable x in the following situations:

1. x appears on the left side of an assignment statement
2. x is an actual parameter in a call site and its value is changed within the method
3. x is a formal parameter of a method (an implicit def when the method begins execution)
4. x is an input to the program

Some features of programming languages greatly complicate this seemingly simple definition. For example, is a def of an array variable a def of the entire array, or of just the element being referenced? What about objects; should the def consider the entire object, or only a particular instance variable of the object? If two variables reference the same location, that is, the variables are aliases, how is the analysis done? What is the relationship between coverage of the original source code, coverage of the optimized source code, and coverage of the machine code? We omit these complicating issues in our presentation and refer advanced readers to the bibliographic notes.

If a variable has multiple definitions in a single basic block, the last definition is the only one that is relevant to data flow analysis.

A use may occur for variable x in the following situations:

1. x appears on the right side of an assignment statement
2. x appears in a conditional test (note that such a test is always associated with at least two edges)
3. x is an actual parameter to a method
4. x is an output of the program
5. x is an output of a method in a return statement or returned as a parameter

Not all uses are relevant for data flow analysis. Consider the following statements that reference local variables (ignoring concurrency):

```
y = z;
x = y + 2;
```

The use of y in the second statement is called a *local* use; it is impossible for a def in another basic block to reach the use in x = y + 2. The reason is that the definition of y in y = z; *always* overrides any definition of y from any other basic block. That is, no def-clear path goes from any other def to that use. In contrast, the use of z is called *global*, because the definition of z used in this basic block must originate in some other basic block. Data flow analysis only considers global uses.

The TestPat example in Figure 2.21 illustrates dataflow analysis for a simple string pattern matching program called TestPat written in Java.

The CFG for TestPat was previously shown in Figure 2.12, with the actual Java statements annotated on the nodes and edges.

The CFG for TestPat with def and use sets explicitly marked was shown in Figure 2.13. While numerous tools can create CFGs for programs, it helps students to

```
// Example program for pattern matching of two strings
class TestPat
{
  public static void main (String[] argv)
  {
    final int MAX = 100;
    char subject[] = new char[MAX];
    char pattern[] = new char[MAX];
    if (argv.length != 2)
    {
      System.out.println
      ("java TestPat String-Subject String-Pattern");
      return;
    }
    subject = argv[0].toCharArray();
    pattern = argv[1].toCharArray();
    TestPat testPat = new TestPat ();
    int n = 0;
    if ((n = testPat.pat (subject, pattern)) == -1)
      System.out.println
      ("Pattern string is not a substring of the subject string");
    else
      System.out.println
      ("Pattern string begins at the character " + n);
  }

  public TestPat ()
  { }

  public int pat (char[] subject, char[] pattern)
  {
  // Post: if pattern is not a substring of subject, return -1
  //       else return (zero-based) index where the pattern (first)
  //       starts in subject

    final int NOTFOUND = -1;
    int  iSub = 0, rtnIndex = NOTFOUND;
    boolean isPat  = false;
    int subjectLen = subject.length;
    int patternLen = pattern.length;

    while (isPat == false && iSub + patternLen - 1 < subjectLen)
    {
      if (subject [iSub] == pattern [0])
      {
        rtnIndex = iSub; // Starting at zero
        isPat = true;
        for (int iPat = 1; iPat < patternLen; iPat ++)
        {
          if (subject[iSub + iPat] != pattern[iPat])
          {
            rtnIndex = NOTFOUND;
            isPat = false;
            break;  // out of for loop
          }
        }
      }
      iSub ++;
    }
    return (rtnIndex);
  }
}
```

Figure 2.21. TestPat for data flow example.

create CFGs by hand. When doing so, a good habit is to draw the CFG first with the statements, then redraw it with the def and use sets.

Table 2.1 lists the defs and uses at each node in the CFG for TestPat This simply repeats the information in Figure 2.13, but in a convenient form. Table 2.2 contains the same information for edges. We suggest that beginning students check their

Table 2.1. Defs and uses at each node in the CFG for TestPat

node	def	use
1	{subject, pattern}	
2	{NOTFOUND, isPat, iSub, rtnIndex, subjectLen, patternLen}	{subject, pattern}
3		
4		
5	{rtnIndex, isPat, iPat}	{iSub}
6		
7		
8	{rtnIndex, isPat}	{NOTFOUND}
9	{iPat}	{iPat}
10	{iSub}	{iSub}
11		{rtnIndex}

understanding of these definitions by verifying that the contents of these two tables are correct.

Finally, we list the du-paths for each variable in TestPat followed by all the du-paths for each du-pair in Table 2.3. The first column gives the variable name, and the second gives the def node number and variable (that is, the left side of the formula that lists all the du-paths with respect to the variable, as defined in Section 2.2.2). The third column lists all the du-paths that start with that def. If a du-pair has more than one path to the same use, they are listed on multiple rows with subpaths that end with the same node number. The fourth column, "prefix?", is a notational convenience that is explained below. This information is extremely tedious to derive by hand, and testers tend to make many errors. This analysis is best done automatically.

Several def/use pairs have more than one du-path in TestPat. For example, the variable iSub is defined in node 2 and used in node 10. Three du-paths exist, [2,3,4,10](iSub), [2,3,4,5,6,10](iSub), and [2,3,4,5,6,7,8,10](iSub).

Table 2.2. Defs and uses at each edge in the CFG for TestPat.

edge	use
(1, 2)	
(2, 3)	
(3, 4)	{iSub, patternLen, subjectLen, isPat}
(3, 11)	{iSub, patternLen, subjectLen, isPat}
(4, 5)	{subject, iSub, pattern}
(4, 10)	{subject, iSub, pattern}
(5, 6)	
(6, 7)	{iPat, patternLen}
(6, 10)	{iPat, patternLen}
(7, 8)	{subject, iSub, iPat, pattern}
(7, 9)	{subject, iSub, iPat, pattern}
(8, 10)	
(9, 6)	
(10, 3)	

Table 2.3. Du-path sets for each variable in TestPat

variable	du-path set	du-paths	prefix?
NOTFOUND	du (2, NOTFOUND)	[2,3,4,5,6,7,8]	
rtnIndex	du (2, rtnIndex)	[2,3,11]	
	du (5, rtnIndex)	[5,6,10,3,11]	
	du (8, rtnIndex)	[8,10,3,11]	
iSub	du (2, iSub)	[2,3,4]	Yes
		[2,3,4,5]	Yes
		[2,3,4,5,6,7,8]	Yes
		[2,3,4,5,6,7,9]	
		[2,3,4,5,6,10]	
		[2,3,4,5,6,7,8,10]	
		[2,3,4,10]	
		[2,3,11]	
	du (10, iSub)	[10,3,4]	Yes
		[10,3,4,5]	Yes
		[10,3,4,5,6,7,8]	Yes
		[10,3,4,5,6,7,9]	
		[10,3,4,5,6,10]	
		[10,3,4,5,6,7,8,10]	
		[10,3,4,10]	
		[10,3,11]	
iPat	du (5, iPat)	[5,6,7]	Yes
		[5,6,10]	
		[5,6,7,8]	
		[5,6,7,9]	
	du (9, iPat)	[9,6,7]	Yes
		[9,6,10]	
		[9,6,7,8]	
		[9,6,7,9]	
isPat	du (2, isPat)	[2,3,4]	
		[2,3,11]	
	du (5, isPat)	[5,6,10,3,4]	
		[5,6,10,3,11]	
	du (8, isPat)	[8,10,3,4]	
		[8,10,3,11]	
subject	du (1, subject)	[1,2]	Yes
		[1,2,3,4,5]	Yes
		[1,2,3,4,10]	
		[1,2,3,4,5,6,7,8]	
		[1,2,3,4,5,6,7,9]	
pattern	du (1, pattern)	[1,2]	Yes
		[1,2,3,4,5]	Yes
		[1,2,3,4,10]	
		[1,2,3,4,5,6,7,8]	
		[1,2,3,4,5,6,7,9]	
subjectLen	du (2, subjectLen)	[2,3,4]	
		[2,3,11]	
patternLen	du (2, patternLen)	[2,3,4]	Yes
		[2,3,11]	
		[2,3,4,5,6,7]	
		[2,3,4,5,6,10]	

Table 2.4. Test paths to satisfy all du-paths coverage on TestPat

test case (subject,pattern,output)	test path(t)
(a, bc, −1)	[1,2,3,11]
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]
(ab, ac, −1)	[1,2,3,4,5,6,7,8,10,3,11]
(ab, b, 1)	[1,2,3,4,10,3,4,5,6,10,3,11]
(ab, c, −1)	[1,2,3,4,10,3,4,10,3,11]
(abc, abc, 0)	[1,2,3,4,5,6,7,9,6,7,9,6,10,3,11]
(abc, abd, −1)	[1,2,3,4,5,6,7,9,6,7,8,10,3,11]
(abc, ac −1)	[1,2,3,4,5,6,7,8,10,3,4,10,3,11]
(abc, ba, −1)	[1,2,3,4,10,3,4,5,6,7,8,10,3,11]
(abc, bc, 1)	[1,2,3,4,10,3,4,5,6,7,9,6,10,3,11]

One optimization uses the fact that a du-path must be toured by any test that tours an extension of that du-path. These du-paths are marked with the annotation "Yes" in the prefix? column of the table. For example, [2,3,4](iSub) is necessarily toured by any test that tours the du-path [2,3,4,5,6,7,8](iSub), because [2,3,4] is a prefix of [2,3,4,5,6,7,8]. Thus, the path is not considered in the subsequent table that relates du-paths to test paths that tour them. One has to be a bit careful with this optimization, since the extended du-path may be infeasible even if the prefix is not.

Table 2.4 shows that a relatively small set of 11 test cases satisfies all du-paths coverage on this example. (One du-path is infeasible.) The reader may wish to evaluate this test set with the non-data flow graph coverage criteria.

Table 2.5 lists which du-paths are toured by each test case. For each test case in the first column, the test path that is executed by that test is shown in the second column, and the du-path that is toured by the test path is shown in the third column.

Table 2.5. Test paths and du-paths covered on TestPat.

test case (subject,pattern, output)	test path(t)	du-path toured
(ab, ac, −1)	[1,2,3,4,5,6,7,8,10,3,11]	[2,3,4,5,6,7,8](NOTFOUND)
(a, bc, −1)	[1,2,3,11]	[2,3,11](rtnIndex)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[5,6,10,3,11](rtnIndex)
(ab, ac, −1)	[1,2,3,4,5,6,7,8,10,3,11]	[8,10,3,11](rtnIndex)
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]	[2,3,4,5,6,7,9] (iSub)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[2,3,4,5,6,10](iSub)
(ab, ac, −1)	[1,2,3,4,5,6,7,8,10,3,11]	[2,3,4,5,6,7,8,10](iSub)
(ab, c, −1)	[1,2,3,4,10,3,4,10,3,11]	[2,3,4,10](iSub)
(a, bc, −1)	[1,2,3,11]	[2,3,11] (iSub)
(abc, bc, 1)	[1,2,3,4,10,3,4,5,6,7,9,6,10,3,11]	[10,3,4,5,6,7,9](iSub)
(ab, b, 1)	[1,2,3,4,10,3,4,5,6,10,3,11]	[10,3,4,5,6,10](iSub)
(abc, ba, −1)	[1,2,3,4,10,3,4,5,6,7,8,10,3,11]	[10,3,4,5,6,7,8,10](iSub)
(ab, c, −1)	[1,2,3,4,10,3,4,10,3,11]	[10,3,4,10](iSub)

Table 2.5. *Continued*

test case (subject,pattern, output)	test path(t)	du-path toured
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[10,3,11](iSub)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[5,6,10](iPat)
(ab, ac, −1)	[1,2,3,4,5,6,7,8,10,3,11]	[5,6,7,8](iPat)
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]	[5,6,7,9](iPat)
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]	[9,6,10](iPat)
(abc, abd, −1)	[1,2,3,4,5,6,7,9,6,7,8,10,3,11]	[9,6,7,8](iPat)
(abc, abc, 0)	[1,2,3,4,5,6,7,9,6,7,9,6,10,3,11]	[9,6,7,9](iPat)
(ab, ac, −1)	[1,2,3,4,5,6,7,8,10,3,11]	[2,3,4](isPat)
(a, bc, −1)	[1,2,3,11]	[2,3,11](isPat)
No test case	Infeasible	[5,6,10,3,4](isPat)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[5,6,10,3,11](isPat)
(abc, ac −1)	[1,2,3,4,5,6,7,8,10,3,4,10,3,11]	[8,10,3,4](isPat)
(ab, ac, −1)	[1,2,3,4,5,6,7,8,10,3,11]	[8,10,3,11](isPat)
(ab, c, −1)	[1,2,3,4,10,3,4,10,3,11]	[1,2,3,4,10](subject)
(ab, ac, −1)	[1,2,3,4,5,6,7,8,10,3,11]	[1,2,3,4,5,6,7,8](subject)
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]	[1,2,3,4,5,6,7,9](subject)
(ab, c, −1)	[1,2,3,4,10,3,4,10,3,11]	[1,2,3,4,10](pattern)
(ab, ac, −1)	[1,2,3,4,5,6,7,8,10,3,11]	[1,2,3,4,5,6,7,8](pattern)
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]	[1,2,3,4,5,6,7,9](pattern)
(ab, c, −1)	[1,2,3,4,10,3,4,10,3,11]	[2,3,4](subjectLen)
(a, bc, −1)	[1,2,3,11]	[2,3,11](subjectLen)
(a, bc, −1)	[1,2,3,11]	[2,3,11](patternLen)
(ab, ac, −1)	[1,2,3,4,5,6,7,8,10,3,11]	[2,3,4,5,6,7](patternLen)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[2,3,4,5,6,10](patternLen)

EXERCISES

Section 2.3.

1. Use the following program fragment for questions (a)–(e) below.

```
w = x;          // node 1
if (m > 0)
{
  w++;          // node 2
}
else
{
  w=2*w;        // node 3
}
// node 4 (no executable statement)
if (y <= 10)
{
  x = 5*y;      // node 5
}
else
```

```
{
  x = 3*y+5;  // node 6
}
z = w + x;    // node 7
```

(a) Draw a control flow graph for this program fragment. Use the node numbers given above.
(b) Which nodes have defs for variable w?
(c) Which nodes have uses for variable w?
(d) Are there any du-paths with respect to variable w from node 1 to node 7? If not, explain why not. If any exist, show one.
(e) Enumerate all of the du-paths for variables w and x.

2. Select a commercial coverage tool of your choice. Note that some have free trial evaluations. Choose a tool, download it, and run it on some software. You can use one of the examples from this text, software from your work environment, or software available over the Web. Write up a short summary report of your experience with the tool. Be sure to include any problems installing or using the tool. The main grading criterion is that you actually collect some coverage data for a reasonable set of tests on some program.
3. Consider the pattern matching example in Figure 2.21. Instrument the code so as to be able to produce the execution paths reported in the text for this example. That is, on a given test execution, your instrumentation program should compute and print the corresponding test path. Run the instrumented program on the test cases listed at the end of Section 2.3.
4. Consider the pattern matching example in Figure 2.21. In particular, consider the final table of tests in Section 2.3. Consider the variable *iSub*. Number the (unique) test cases, starting at 1, from the top of the *iSub* part of the table. For example, $(ab, c, -1)$, which appears twice in the *iSub* portion of the table, should be labeled test $t4$.
   (a) Give a minimal test set that satisfies *all defs* coverage. Use the test cases given.
   (b) Give a minimal test set that satisfies *all uses* coverage.
   (c) Give a minimal test set that satisfies *all du-paths* coverage.
5. Again consider the pattern matching example in Figure 2.21. Instrument the code so as to produce the execution paths reported in the text for this example. That is, on a given test execution, your tool should compute and print the corresponding test path. Run the following three test cases and answer questions (a)–(g) below:
   - *subject* = "brown owl" *pattern* = "wl" *expected output* = 7
   - *subject* = "brown fox" *pattern* = "dog" *expected output* = −1
   - *subject* = "fox" *pattern* = "brown" *expected output* = −1

   (a) Find the actual path followed by each test case.
   (b) For each path, give the du-paths that the path tours in the table at the end of Section 2.3. To reduce the scope of this exercise, consider only the following du-paths: *du (10, iSub)*, *du (2, isPat)*, *du (5, isPat)*, and *du (8, isPat)*.
   (c) Explain why the du-path [5, 6, 10, 3, 4] cannot be toured by any test path.

(d) Select tests from the table at the end of Section 2.3 to complete coverage of the (feasible) du-paths that are uncovered in question a.
(e) From the tests above, find a minimal set of tests that achieves All-Defs coverage with respect to the variable *isPat*.
(f) From the tests above, find a minimal set of tests that achieves All-Uses Coverage with respect to the variable *isPat*.
(g) Is there any difference between All-Uses coverage and all du-paths coverage with respect to the variable *isPat* in the *pat* method?

6. Use the following method **fmtRewrap()** for questions a–e below.

```
 1. /** ****************************************************
 2.  *  Rewraps the string (Similar to the Unix fmt).
 3.  *  Given a string S, eliminate existing CRs and add CRs to the
 4.  *  closest spaces before column N.  Two CRs in a row are considered to
 5.  *  be "hard CRs" and are left alone.
 7. *******************************************************/
 6.
 8. static final char CR = '\n';
 9. static final int inWord        = 0;
10. static final int betweenWord = 1;
11. static final int lineBreak      = 2;
12. static final int crFound        = 3;
13. static private String fmtRewrap (String S, int N)
14. {
15.    int state = betweenWord;
16.    int lastSpace = -1;
17.    int col = 1;
18.    int i = 0;
19.    char c;
20.
21.    char SArr [] = S.toCharArray();
22.    while (i < S.length())
23.    {
24.       c = SArr[i];
25.       col++;
26.       if (col >= N)
27.          state = lineBreak;
28.       else if (c == CR)
29.          state = crFound;
30.       else if (c == ' ')
31.          state = betweenWord;
32.       else
33.          state = inWord;
34.       switch (state)
35.       {
36.       case betweenWord:
```

```
37.         lastSpace = i;
38.         break;
39.
40.       case lineBreak:
41.         SArr [lastSpace] = CR;
42.         col = i-lastSpace;
43.         break;
44.
45.       case crFound:
46.         if (i+1 < S.length() && SArr[i+1] == CR)
47.         {
48.           i++; // Two CRs => hard return
49.           col = 1;
50.         }
51.         else
52.           SArr[i] = '';
53.         break;
54.
55.       case inWord:
56.       default:
57.         break;
58.       } // end switch
59.       i++;
60.     } // end while
61.     S = new String (SArr) + CR;
62.     return (S);
63. }
```

(a) Draw the control flow graph for the **fmtRewrap()** method.
(b) For **fmtRewrap()**, find a test case such that the corresponding test path visits the edge that connects the beginning of the **while** statement to the **S = new String(SArr) + CR;** statement **without** going through the body of the while loop.
(c) Enumerate the test requirements for node coverage, edge coverage, and prime path coverage for the graph for **fmtRewrap()**.
(d) List test paths that achieve node coverage but not edge coverage on the graph.
(e) List test paths that achieve edge coverage but not prime path coverage on the graph.

7. Use the following method **printPrimes()** for questions a–f below.

```
1. /** ****************************************************
2.  * Finds and prints n prime integers
3.  * Jeff Offutt, Spring 2003
4. ******************************************************** */
5. private static void printPrimes (int n)
6. {
```

```
int curPrime;          // Value currently considered for primeness
int numPrimes;         // Number of primes found so far.
boolean isPrime;       // Is curPrime prime?
int [] primes = new int [MAXPRIMES]; // The list of prime numbers.

// Initialize 2 into the list of primes.
primes [0] = 2;
numPrimes = 1;
curPrime  = 2;
while (numPrimes < n)
{
  curPrime++;  // next number to consider ...
  isPrime = true;
  for (int i = 0; i <= numPrimes-1; i++)
   {  // for each previous prime.
    if (isDivisible (primes[i], curPrime))
    {  // Found a divisor, curPrime is not prime.
      isPrime = false;
      break; // out of loop through primes.
    }
  }
  if (isPrime)
  {  // save it!
    primes[numPrimes] = curPrime;
    numPrimes++;
  }
}  // End while

// Print all the primes out.
for (int i = 0; i <= numPrimes-1; i++)
{
  System.out.println ("Prime: " + primes[i]);
}
}  // end printPrimes
```

(a) Draw the control flow graph for the **printPrimes()** method.

(b) Consider test cases $t1 = (n = 3)$ and $t2 = (n = 5)$. Although these tour the same prime paths in *printPrimes()*, they do not necessarily find the same faults. Design a simple fault that $t2$ would be more likely to discover than $t1$ would.

(c) For *printPrimes*(), find a test case such that the corresponding test path visits the edge that connects the beginning of the **while** statement to the **for** statement **without** going through the body of the while loop.

(d) Enumerate the test requirements for node coverage, edge coverage, and prime path coverage for the graph for *printPrimes*().

(e) List test paths that achieve node coverage but not edge coverage on the graph.
(f) List test paths that achieve edge coverage but not prime path coverage on the graph.

2.4 GRAPH COVERAGE FOR DESIGN ELEMENTS

Use of data abstraction and object-oriented software has led to an increased emphasis on modularity and reuse. This means that testing of software based on various parts of the design (*design elements*) is becoming more important than in the past. These activities are usually associated with integration testing. One of the benefits of modularity is that the software components can be tested independently, which is usually done by programmers during unit and module testing.

2.4.1 Structural Graph Coverage for Design Elements

Graph coverage for design elements usually starts by creating graphs that are based on couplings between software components. *Coupling* measures the dependency relations between two units by reflecting their interconnections; faults in one unit may affect the coupled unit. Coupling provides summary information about the design and the structure of the software. Most test criteria for design elements require that various connections among program components be visited.

The most common graph used for structural design coverage is the *call graph*. In a call graph, the nodes represent methods (or units) and the edges represent method calls. Figure 2.22 represents a small program that contains six methods. Method `A` calls `B`, `C`, and `D`, and `C` in turn calls `E` and `F`, and `D` also calls `F`.

The coverage criteria from Section 2.2.1 can be applied to call graphs. Node coverage requires that each method be called at least once and is also called *method coverage*. Edge coverage requires that each call be executed at least once and is also called *call coverage*. For the example in Figure 2.22, node coverage requires

Figure 2.22. A simple call graph.

that each method be called at least once, whereas edge coverage requires that F be called at least twice, once from C and once from D.

Application to Modules

Recall from Chapter 1 that a module is a collection of related units, for example a class is Java's version of a module. As opposed to complete programs, the units in a class may not all call each other. Thus, instead of being able to obtain one connected call graph, we may generate several disconnected call graphs. In a simple degenerative case (such as for a simple stack), there may be no calls between units. In these cases, module testing with this technique is not appropriate. Techniques based on sequences of calls are needed.

Inheritance and Polymorphism

The object-oriented language features of inheritance and polymorphism introduce new abilities for designers and programmers, but also new problems for testers. As of this writing, it is still not clear how best to test these language features or what criteria are appropriate. This text introduces the current state of knowledge; the interested reader is encouraged to keep up with the literature for continuing results and techniques for testing OO software. The bibliographic notes give some current references, and further ideas are discussed in Chapter 7. The most obvious graph to create for testing these features (which we collectively call "the OO language features") is the inheritance hierarchy. Figure 2.23 represents a small inheritance hierarchy with four classes. Classes C and D inherit from B, and B in turn inherits from A.

The coverage criteria from Section 2.2.1 can be applied to inheritance hierarchies in ways that are superficially simple, but have some subtle problems. In OO programming, classes are not directly tested because they are not executable. In fact, the edges in the inheritance hierarchy do not represent execution flow at all, but rather inheritance dependencies. To apply any type of coverage, we first need a model for

Figure 2.23. A simple inheritance hierarchy.

Figure 2.24. An inheritance hierarchy with objects instantiated.

what coverage means. The first step is to require that objects be instantiated for some or all of the classes. Figure 2.24 shows the inheritance hierarchy from Figure 2.23 with one object instantiated for each class.

The most obvious interpretation of node coverage for this graph is to require that at least one object be created for each class. However, this seems weak because it says nothing about execution. The logical extension is to require that for each object of each class, the call graph must be covered according to the call coverage criterion above. Thus, the *OO call coverage* criterion can be called an "aggregation criterion" because it requires call coverage to be applied on at least one object for each class.

An extension of this is the *all object call* criterion, which requires that call coverage be satisfied for every object that is instantiated for every class.

2.4.2 Data Flow Graph Coverage for Design Elements

Control connections among design elements are simple and straightforward and tests based on them are probably not very effective at finding faults. On the other hand, data flow connections are often very complex and difficult to analyze. For a tester, that should immediately suggest that they are a rich source for software faults. The primary issue is where the defs and uses occur. When testing program units, the defs and uses are in the same unit. During integration testing, defs and uses are in different units. This section starts with some standard compiler/program analysis terms.

A *caller* is a unit that invokes another unit, the *callee*. The statement that makes the call is the *call site*. An *actual parameter* is in the caller; its value is assigned to a *formal parameter* in the callee. The *interface* between two units is the mapping of actual to formal parameters.

The underlying premise of the data flow testing criteria for design elements is that to achieve confidence in the interfaces between integrated program units, it must be ensured that variables defined in caller units be appropriately used in callee

Figure 2.25. An example of parameter coupling.

units. This technique can be limited to the unit interfaces, allowing us to restrict our attention to the **last** definitions of variables just **before** calls to and returns from the called units, and the **first** uses of variables just **after** calls to and returns from the called unit.

Figure 2.25 illustrates the relationships that the data flow criteria will test. The criteria require execution from definitions of actual parameters through calls to uses of formal parameters.

Three types of data flow couplings have been identified. The most obvious is *parameter coupling*, where parameters are passed in calls. *Shared data coupling* occurs when two units access the same data object as a global or other non-local variable, and *external device coupling* occurs when two units access the same external medium such as a file. In the following, all examples and discussion will be in terms of parameters and it will be understood that the concepts apply equally to shared data and external device coupling. We use the general term *coupling variable* for variables that are defined in one unit and used in another.

This form of data flow is concerned only with last-defs before calls and returns and first-uses after calls and returns. That is, it is concerned only with defs and uses immediately surrounding the calls between methods. The last-defs before a call are locations with defs that reach uses at callsites and the last-defs before a return are locations with defs that reach a return statement. The following definitions assume a variable that is defined in either the caller or the callee, and used in the other.

> *Definition 2.40 Last-def:* The set of nodes that define a variable `x` for which there is a def-clear path from the node through the call site to a use in the other unit.

The variable can be passed as a parameter, a return value, or a shared variable reference. If the function has no return statement, an implicit return statement is assumed to exist at the last statement in the method.

The definition for first-use is complementary to that of last-def. It depends on paths that are not only def-clear, but also use-clear. A path from n_i to n_j is *use-clear* with respect to variable v if for every node n_k on the path, $k \neq i$ and $k \neq j$, v is not in $use(n_k)$. Assume that the variable y is used in one of the units after having been defined in the other. Assume that a variable y has received a value that has been

Figure 2.26. Coupling du-pairs.

passed from the other unit, either through parameter passing, a return statement, shared data, or other value passing.

> *Definition 2.41 First-use:* The set of nodes that have uses of *y* and for which there exists a path that is def-clear and use-clear from the entry point (if the use is in the callee) or the call site (if the use is in the caller) to the nodes.

Figure 2.26 shows a caller `F()` and a callee `G()`. The callsite has two du-pairs; `x` in `F()` is passed to `a` in `G()` and `b` in `G()` is returned and assigned to `y` in `F()`. Note that the assignment to `y` in `F()` is explicitly **not** the use, but considered to be part of the transfer. Its use is further down, in the `print(y)` statement.

This definition allows for one anomaly when a return value is not explicitly assigned to a variable, as in the statement `print (f(x))`. In this case, an implicit assignment is assumed and the first-use is in the `print(y)` statement.

Figure 2.27 illustrates last-defs and first-uses between two units with two partial CFGs. The unit on the left, the caller, calls the callee `B`, with one actual parameter, `X`, which is assigned to formal parameter `y`. `X` is defined at nodes 1, 2 and 3, but the def at node 1 cannot reach the call site at node 4, thus the last-defs for `X` is the set {2, 3}. The formal parameter `y` is used at nodes 11, 12, and 13, but no use-clear path goes from the entry point at node 10 to 13, so the first-uses for `y` is the set {11, 12}.

Figure 2.27. Last-defs and first-uses.

Recall that a du-path is a path from a def to a use in the same graph. This notion is refined to a *coupling du-path* with respect to a coupling variable x. A coupling du-path is a path from a last-def to a first-use.

The coverage criteria from Section 2.2.2 can now be applied to coupling graphs. All-Defs coverage requires that a path be executed from every last-def to at least one first-use. In this context, all-defs is called *All-Coupling-Def* coverage. All-Uses coverage requires that a path be executed from every last-def to every first-use. In this context, All-Uses is also called *All-Coupling-Use* coverage.

Finally, All-du-Paths coverage requires that we tour every simple path from every last-def to every first-use. As before, the All-du-Paths criterion can be satisfied by tours that include sidetrips. In this context, All-du-Paths is also called *All-Coupling-du-Paths* coverage.

Example

Now we will turn to an example to illustrate coupling data flow. Class `Quadratic` in Figure 2.28 computes the quadratic root of an equation, given three integer coefficients. The call to `Root()` on line 34 in `main` passes in three parameters. Each of the variables `X`, `Y`, and `Z` have three last-defs in the caller at lines 16, 17, 18, lines 23, 24, and 25, and lines 30, 31, and 32. They are mapped to formal parameters `A`, `B`, and `C` in `Root()`. All three variables have a first-use at line 47. The class variables `Root1` and `Root2` are defined in the callee and used in the caller. Their last-defs are at lines 53 and 54 and the first-use is at line 37.

The value of local variable `Result` is returned to the caller, with two possible last-defs at lines 50 and 55 and first-use at line 35.

The coupling du-pairs can be listed using pairs of triples. Each triple gives a unit name, variable name, and a line number. The first triple in a pair says where the variable is defined, and the second where it is used. The complete set of coupling du-pairs for class `Quadratic` is

```
(main(), X, 16) -- (Root(), A, 47)
(main(), Y, 17) -- (Root(), B, 47)
(main(), Z, 18) -- (Root(), C, 47)
(main(), X, 23) -- (Root(), A, 47)
(main(), Y, 24) -- (Root(), B, 47)
(main(), Z, 25) -- (Root(), C, 47)
(main(), X, 30) -- (Root(), A, 47)
(main(), Y, 31) -- (Root(), B, 47)
(main(), Z, 32) -- (Root(), C, 47)
(Root(), Root1, 53) -- (main(), Root1, 37)
(Root(), Root2, 54) -- (main(), Root2, 37)
(Root(), Result, 50) -- (main(), ok, 35)
(Root(), Result, 55) -- (main(), ok, 35)
```

A couple of notes are important to remember about coupling data flow. First, only variables that are used or defined in the callee are considered. That is, last-defs that have no corresponding first-uses are not useful for testing. Second, we must

```
// Program to compute the quadratic root for two numbers
import java.lang.Math;

class Quadratic
{
private static double Root1, Root2;

public static void main (String[] argv)
{
   int X, Y, Z;
   boolean ok;
   if (argv.length == 3)
   {
      try
      {
         X = Integer.parseInt (argv[1]);
         Y = Integer.parseInt (argv[2]);
         Z = Integer.parseInt (argv[3]);
      }
      catch (NumberFormatException e)
      {
         System.out.println ("Inputs not integers, using 8, 10, -33.");
         X = 8;
         Y = 10;
         Z = -33;
      }
   }
   else
   {
      X = 8;
      Y = 10;
      Z = -33;
   }
   ok = Root (X, Y, Z);
   if (ok)
      System.out.println
         ("Quadratic: Root 1 = " + Root1 + ", Root 2 = " + Root2);
   else
      System.out.println ("No solution.");
}

// Finds the quadratic root, A must be non-zero
private static boolean Root (int A, int B, int C)
{
   double D;
   boolean Result;
   D = (double)(B*B) - (double)(4.0*A*C);
   if (D < 0.0)
   {
      Result = false;
      return (Result);
   }
   Root1 = (double) ((-B + Math.sqrt(D)) / (2.0*A));
   Root2 = (double) ((-B - Math.sqrt(D)) / (2.0*A));
   Result = true;
   return (Result);
} // End method Root

} // End class Quadratic
```

Figure 2.28. Quadratic root program.

remember implicit initialization of class and global variables. In some languages (such as Java and C), class and instance variables are given default values. These definitions can be modeled as occurring at the beginning of appropriate units. For example, class-level initializations may be considered to occur in the main() method or in constructors. Although other methods that access class variables may use the default values on the first call, it is also possible for such methods to use values written by other methods, and hence the normal coupling data flow analysis methods should

Figure 2.29. Def-use pairs under intra-procedural and inter-procedural data flow.

be employed. Also, this analysis is specifically not considering "transitive du-pairs." That is, if unit A calls B, and B calls C, last-defs in A do **not** reach first-uses in C. This type of analysis is prohibitively expensive with current technologies and of questionable value. Finally, data flow testing has traditionally taken an abstract view of array references. Identifying and keeping track of individual array references is an undecidable problem in general and very expensive even in finite cases. So, most tools consider a reference to one element of an array to be a reference to the entire array.

Inheritance and Polymorphism (*Advanced topic*)

The previous discussion covers the most commonly used form of data flow testing as applied beyond the method level. However, the flow of data along couplings between callers and callees is only one type of a very complicated set of data definition and use pairs. Consider Figure 2.29, which shows the types of du-pairs discussed so far. On the left is a method, `A()`, which contains a def and a use. (For this discussion we will omit the variable and assume that all du-pairs refer to the same variable.) The right illustrates that there are two types of inter-procedural du-pairs.

Full inter-procedural data flow identifies **all** du-pairs between a caller (`A()`) and a callee (`B()`). *Coupling inter-procedural* data flow is as described in Section 2.4.2; identifying du-pairs between last-defs and first-uses.

Figure 2.30 illustrates du-pairs in object-oriented software. DU pairs are usually based on the *class* or *state* variables defined for the class. The left picture in Figure 2.30 shows the "direct" case for OO du-pairs. A *coupling method*, `F()`, calls

Figure 2.30. Def-use pairs in object-oriented software.

distributed software
data flow

"message" could be HTTP, RMI, or other mechanism.
A() and B() could be in the same class or accessing a persistent variable such as in a Web session.

Figure 2.31. Def-use pairs in web applications and other distributed software.

two methods, `A()` and `B()`. `A()` defines a variable and `B()` uses it. For the variable reference to be the same, both `A()` and `B()` must be called through the same *instance context*, or object reference. That is, if the calls are `o.A()` and `o.B()`, they are called through the instance context of `o`. If the calls are not made through the same instance context, the definition and use will be to different instances of the variable.

The right side of Figure 2.30 illustrates "indirect" du-pairs. In this scenario, the coupling method `F()` calls two methods, `M()` and `N()`, which in turn call two other methods, `A()` and `B()`. The def and use are in `A()` and `B()`, so the reference is indirect. The analysis for indirect du-pairs is considerably more complicated than for direct du-pairs. It should be obvious that there can be more than one call between the coupling method and the methods with the def and use.

In OO data flow testing, the methods `A()` and `B()` could be in the same class, or they could be in different classes and accessing the same global variables.

Finally, Figure 2.31 illustrates du-pairs in distributed software. `P1` and `P2` are two processes, threads, or other distributed software components, and they call `A()` and `B()`, which def and use the same variable. The distribution and communication could use any of a number of methods, including HTTP (Web-based), remote method invocation (RMI), or CORBA. `A()` and `B()` could be in the same class or could access a persistent variable such as a Web session variable or permanent data store. While this sort of "very loosely coupled" software can be expected to have far fewer du-pairs, identifying them, finding def-clear paths between them, and designing test cases to cover them is quite complicated.

EXERCISES

Section 2.4.

1. Use the class *Stutter* in Figures 2.34 and 2.35 in Section 2.5 to answer questions a–d below.
   (a) Draw a control flow graph for Stutter.
   (b) List all the call sites.
   (c) List all du-pairs for each call site.
   (d) Create test data to satisfy *All-Coupling-Use coverage* for Stutter.

2. Use the following program fragment for questions (a)–(e) below.

```
public static void f1 (int x, int y)
{
  if (x < y) { f2 (y); } else { f3 (y); };
}
public static void f2 (int a)
{
  if (a % 2 == 0) { f3 (2*a); };
}
public static void f3 (int b)
{
  if (b > 0) { f4(); } else { f5(); };
}
public static void f4() {... f6()....}
public static void f5() {... f6()....}
public static void f6() {...}
```

Use the following test inputs:

- $t1 = f1\ (0,\ 0)$
- $t2 = f1\ (1,\ 1)$
- $t3 = f1\ (0,\ 1)$
- $t4 = f1\ (3,\ 2)$
- $t5 = f1\ (3,\ 4)$

(a) Draw the call graph for this program fragment.
(b) Give the path in the graph followed by each test.
(c) Find a minimal test set that achieves node coverage.
(d) Find a minimal test set that achieves edge coverage.
(e) Give the (maximal) prime paths in the graph. Give the maximal prime path that is not covered by any of the test paths above.

3. Use the following methods **trash()** and **takeOut()** to answer questions (a)–(c).

```
1 public void trash (int x)
2 {
3   int m, n;
4
5   m = 0;
6   if (x > 0)
7     m = 4;
8   if (x > 5)
9     n = 3*m;
10  else
11    n = 4*m;
12  int o = takeOut (m, n);
13  System.out.println ("o is: " + o);
14 }
```

```
15 public int takeOut (int a, int b)
16 {
17   int d, e;
18
19   d = 42*a;
20   if (a > 0)
21     e = 2*b+d;
22   else
23     e = b+d;
24   return (e);
25 }
```

(a) Give all call sites using the line numbers given.
(b) Give all pairs of *last-def*s and *first-use*s.
(c) Provide test inputs that satisfy *all-coupling-uses* (note that **trash()** only has one input).

2.5 GRAPH COVERAGE FOR SPECIFICATIONS

Testers can also use software specifications as sources for graphs. The literature presents many techniques for generating graphs and criteria for covering those graphs, but most of them are in fact very similar. We begin by looking at graphs based on *sequencing constraints* among methods in classes, then graphs that represent state behavior of software.

2.5.1 Testing Sequencing Constraints

We pointed out in Section 2.4.1 that call graphs for classes often wind up being disconnected, and in many cases, such as with small abstract data types (ADTs), methods in a class share no calls at all. However, the order of calls is almost always constrained by rules. For example, many ADTs must be initialized before being used, we cannot pop an element from a stack until something has been pushed onto it, and we cannot remove an element from a queue until an element has been put on it. These rules impose constraints on the order in which methods may be called. Generally, a *sequencing constraint* is a rule that imposes some restriction on the order in which certain methods may be called.

Sequencing constraints are sometimes explicitly expressed, sometimes implicitly expressed, and sometimes not expressed at all. Sometimes they are encoded as a precondition or other specification, but not directly as a sequencing condition. For example, consider the following precondition for `DeQueue()`:

```
public int DeQueue ()
{
// Pre: At least one element must be on the queue.
.
:
public EnQueue (int e)
{
// Post: e is on the end of the queue.
```

Although it is not said explicitly, a wise programmer can infer that the only way an element can "be on the queue" is if `EnQueue()` has previously been called. Thus, an implicit sequencing constraint occurs between `EnQueue()` and `DeQueue()`.

Of course, formal specifications can help make the relationships more precise. A wise tester will certainly use formal specifications when available, but a responsible tester must look for formal relationships even when they are not explicitly stated. Also, note that sequencing constraints do not capture all the behavior, but only abstract certain key aspects. The sequence constraint that `EnQueue()` must be called

Figure 2.32. Control flow graph using the File ADT.

before DeQueue() does not capture the fact that if we only EnQueue() one item, and then try to DeQueue() two items, the queue will be empty. The precondition may capture this fact, but usually not in a formal way that automated tools can use. This kind of relationship is beyond the ability of a simple sequencing constraint but can be dealt with by some of the state behavior techniques in the next section.

This relationship is used in two places during testing. We illustrate them with a small example of a class that encapsulates operations on a file. Our class FileADT will have three methods:

- open (String fName) // Opens the file with the name fName
- close (String fName) // Closes the file and makes it unavailable for use
- write (String textLine) // Writes a line of text to the file

This class has several sequencing constraints. The statements use "must" and "should" in very specific ways. When "must" is used, it implies that violation of the constraint is a fault. When "should" is used, it implies that violation of the constraint is a potential fault, but not necessarily.

1. An open(F) must be executed before every write(t)
2. An open(F) must be executed before every close()
3. A write(t) must not be executed after a close() unless an open(F) appears in between
4. A write(t) should be executed before every close()
5. A close() must not be executed after a close() unless an open(F) appears in between
6. An open(F) must not be executed after an open(F) unless a close() appears in between

Constraints are used in testing in two ways to evaluate software that uses the class (a "client"), based on the CFG of Section 2.3.1. Consider the two (partial) CFGs in Figure 2.32, representing two units that use FileADT. We can use this graph to test the use of the FileADT class by checking for sequence violations. This can be done both statically and dynamically.

Static checks (not considered to be traditional testing) proceed by checking each constraint. First consider the write(t) statements at nodes 2 and 5 in graph (a). We can check to see whether paths exist from the open(F) at node 1 to nodes 2 and 5 (constraint 1). We can also check whether a path exists from the open(F) at node 1 to the close() at node 6 (constraint 2). For constraints 3 and 4, we can check to see if a path goes from the close() at node 6 to any of the write(t) statements, and see if a path exists from the open(F) to the close() that does not go through at least one write(t). This will uncover one possible problem, the path [1, 3, 4, 6] goes from an open(F) to a close() with no intervening write(t) calls.

For constraint 5, we can check if a path exists from a close() to a close() that does not go through an open(F). For constraint 6, we can check if a path exists from an open(F) to an open(F) that does not go through a close().

This process will find a more serious problem with graph (b) in 2.32. A path exists from the close() at node 7 to the write(t) at node 5 and to the write(t) at node 4. While this may seem simple enough not to require formalism for such small graphs, this process is quite difficult with large graphs containing dozens or hundreds of nodes.

Dynamic testing follows a slightly different approach. Consider the problem in graph (a) where no write() appears on the possible path [1, 3, 4, 6]. It is quite possible that the logic of the program dictates that the edge (3, 4) can *never* be taken unless the loop [3, 5, 3] is taken at least once. Because deciding whether the path [1, 3, 4, 6] can be taken or not is formally undecidable, this situation can be checked only by dynamic execution. Thus we generate test requirements to try to *violate* the sequencing constraints. For the FileADT class, we generate the following sets of test requirements:

1. Cover every path from the start node to every node that contains a write(t) such that the path does not go through a node containing an open(F).
2. Cover every path from the start node to every node that contains a close() such that the path does not go through a node containing an open(F).
3. Cover every path from every node that contains a close() to every node that contains a write(t) such that the path does not contain an open(F).
4. Cover every path from every node that contains an open(F) to every node that contains a close() such that the path does not go through a node containing a write(t).
5. Cover every path from every node that contains an open(F) to every node that contains an open(F).

Of course, all of these test requirements will be infeasible in well written programs. However, any tests created as a result of these requirements will almost certainly reveal a fault if one exists.

2.5.2 Testing State Behavior of Software

The other major method for using graphs based on specifications is to model state behavior of the software by developing some form of finite state machine (FSM). Over the last 25 years, many suggestions have been made for creating FSMs and how to test software based on the FSM. The topic of how to create, draw, and interpret a

FSM has filled entire textbooks, and authors have gone into great depth and effort to define what exactly goes into a state, what can go onto edges, and what causes transitions. Rather than using any particular notation, we choose to define a very generic model for FSMs that can be adapted to virtually any notation. These FSMs are essentially graphs, and the graph testing criteria already defined can be used to test software that is based on the FSM.

One of the advantages of basing tests on FSMs is that huge numbers of practical software applications are based on a FSM model or can be modeled as FSMs. Virtually all embedded software fits in this category, including software in remote controls, household appliances, watches, cars, cell phones, airplane flight guidance, traffic signals, railroad control systems, network routers, and factory automation. Indeed, most software can be modeled with FSMs, the primary limitation being the number of states needed to model the software. Word processors, for example, contain so many commands and states that modeling them as FSMs is probably impractical.

Creating FSMs often has great value. If the test engineer creates a FSM to describe existing software, he or she will almost certainly find faults in the software. Some would even argue the converse; if the designers created FSMs, the testers should not bother creating them because problems will be rare.

FSMs can be annotated with different types of actions, including actions on transitions, entry actions on nodes, and exit actions on nodes. Many languages are used to describe FSMs, including UML statecharts, finite automata, state tables (SCR), and petri nets. This book presents examples with basic features that are common to many languages. It is closest to UML statecharts, but not exactly the same.

A *finite state machine* is a graph whose nodes represent states in the execution behavior of the software and edges represent transitions among the states. A *state* represents a recognizable situation that remains in existence over some period of time. A state is defined by specific values for a set of variables; as long as those variables have those values the software is considered to be in that state. (Note that these variables are defined at the design modeling level and may not necessarily correspond to variables in the software.) A *transition* is thought of as occurring in zero time and usually represents a change to the values of one or more variables. When the variables change, the software is considered to move from the transition's *pre-state* (predecessor) to its *post-state* (successor). (If a transition's pre-state and post-state are the same, then values of state variables will not change.) FSMs often define *preconditions* or *guards* on transitions, which define values that specific variables must have for the transition to be enabled, and *triggering events*, which are changes in variable values that cause the transition to be taken. A triggering event "triggers" the change in state. For example, the modeling language SCR calls these WHEN conditions and triggering events. The values the triggering events have before the transition are called *before-values*, and the values after the transition are called *after-values*. When graphs are drawn, transitions are often annotated with the guards and the values that change.

Figure 2.33 illustrates this model with a simple transition that opens an elevator door. If the elevator button is pressed (the triggering event), the door opens only if the elevator is not moving (the precondition, $elevSpeed = 0$).

Given this type of graph, many of the previous criteria can be defined directly. Node coverage requires that each state in the FSM be visited at least once and is

Figure 2.33. Elevator door open transition.

called *state coverage*. Edge coverage is applied by requiring that each transition in the FSM be visited at least once, which is called *transition coverage*. The edge-pair coverage criterion was originally defined for FSMs and is also called *transition-pair* and *two-trip*.

The data flow coverage criteria are a bit more troublesome for FSMs. In most formulations of FSMs, nodes are not allowed to have defs or uses of variables. That is, all of the action is on the transitions. Unlike with code-based graphs, different edges from the same node in a FSM need not have the same set of defs and uses. In addition, the semantics of the triggers imply that the effects of a change to the variables involved are felt immediately by taking a transition to the next state. That is, defs of triggering variables immediately reach uses.

Thus, the All-Defs and All-Uses criteria can only be applied meaningfully to variables involved in guards. This also brings out a more practical problem, which is that the FSMs do not always model assignment to all variables. That is, the uses are clearly marked in the FSM, but defs are not always easy to find. Because of these reasons, few attempts have been made to apply data flow criteria to FSMs.

Deriving Finite State Machine Graphs

One of the difficult parts of applying graph techniques to FSMs is deriving the FSM model of the software in the first place. As we said earlier, FSM models of the software may already exist, or may not. If not, the tester is likely to dramatically increase his or her understanding of the software by deriving the FSMs. However, it is not necessarily obvious how to go about deriving a FSM, so we offer some suggestions. This is not a complete tutorial on constructing FSMs; indeed, a number of complete texts exist on the subject and we recommend that the interested reader study these elsewhere.

This section offers some simple and straightforward suggestions to help readers who are unfamiliar with FSMs get started and avoid some of the more obvious mistakes. We offer the suggestions in terms of a running example, the class Stutter in Figures 2.34 and 2.35. Class Stutter checks each adjacent pair of words in a text file and prints a message if a pair is identical. The second author originally wrote it to edit his papers and find a common mistake mistake.

Class Stutter has a main method and three support methods. When left to their own devices, students will usually pick one of four strategies for generating FSMs from code. Each of these is discussed in turn.

1. Combining control flow graphs
2. Using the software structure
3. Modeling state variables
4. Using the implicit or explicit specifications

1. *Combining control flow graphs*: For programmers who have little or no knowledge of FSMs, this is often the most natural approach to deriving FSMs. Our experience

```
/** ******************************************************
// Stutter checks for repeat words in a text file.
// It prints a list of repeat words, by line number.
// Stutter will accept standard input or a list
// of file names.
//  Jeff Offutt, June 1989 (in C), Java version March 2003
//******************************************************** */
class Stutter
{
  // Class variables used in multiple methods.
  private static boolean lastdelimit = true;
  private static String curWord = "", prevWord = "";
  private static char delimits [] =
        {'', ' ', ' ', ' ', '.', '.', '!', '-', '+', '=', ';', '.', '.', '?',
         '&', '{', '}', '\\'}; // First char in list is a tab

//*************************************************
// main parses the arguments, decides if stdin
// or a file name, and calls Stut().
//*************************************************
public static void main (String[] args) throws IOException
{
  String fileName;
  FileReader myFile;
  BufferedReader inFile = null;

  if (args.length == 0)
  {  // no file, use stdin
    inFile = new BufferedReader (new InputStreamReader (System.in));
  }
  else
  {
    fileName = args [0];
    if (fileName == null)
    {  // no file name, use stdin
      inFile = new BufferedReader (new InputStreamReader (System.in));
    }
    else
    {  // file name, open the file.
      myFile = new FileReader (fileName);
      inFile = new BufferedReader (myFile);
    }
  }

  stut (inFile);
}

//*************************************************
// Stut() reads all lines in the input stream, and
// finds words. Words are defined as being surrounded
// by delimiters as defined in the delimits[] array.
// Every time an end of word is found, checkDupes()
// is called to see if it is the same as the
// previous word.
//*************************************************
private static void stut (BufferedReader inFile) throws IOException
{
  String inLine;
  char c;
  int linecnt = 1;
```

Figure 2.34. Stutter – Part A.

has been that the majority of students will use this approach if not guided away from it. A control flow graph-based FSM for class Stutter is given in Figure 2.36.

The graph in Figure 2.36 is **not** a FSM at all, and this is not the way to form graphs from software. This method has several problems, the first being that the nodes are not states. The methods must return to the appropriate callsites, which means that the graphs contain built-in nondeterminism. For example, in

```
    while ((inLine = inFile.readLine()) != null)
    {  // For each line

      for (int i=0; i<inLine.length(); i++)
      {  // for each character
        c = inLine.charAt(i);

        if (isDelimit (c))
        {  // Found an end of a word.
          checkDupes (linecnt);
        }
        else
        {
          lastdelimit = false;
          curWord = curWord + c;
        }
      }
      linecnt++;
      checkDupes (linecnt);
    }
  }  // end Stut

  //*************************************************
  // checkDupes() checks to see if the globally defined
  // curWord is the same as prevWord and prints a message
  // if they are the same.
  //*************************************************
  private static void checkDupes (int line)
  {
    if (lastdelimit)
      return; // already checked, keep skipping
    lastdelimit = true;
    if (curWord.equals(prevWord))
    {
      System.out.println ("Repeated word on line " + line + ": " +
                          prevWord+ " " + curWord);
    }
    else
    {
      prevWord = curWord;
    }
    curWord = "";
  }  // end checkDupes

  //*************************************************
  // Checks to see if a character is a delimiter.
  //*************************************************
  private static boolean isDelimit (char C)
  {
    for (int i = 0; i < delimits.length; i++)
      if (C == delimits [i])
        return (true);
    return (false);
  }

  } // end class Stutter
```

Figure 2.35. Stutter – Part B.

Figure 2.36, there is an edge from node 2 in checkDupes() to node 6 in stut(), and also an edge from node 2 in checkDupes() to node 8 in stut(). Which edge is taken depends on whether the edge from node 6 in stut() was taken to enter checkDupes() or the edge from node 8 in stut() was taken to enter checkDupes(). Second, the implementation must be finished before the graph can be built; remember from Chapter 1 that one of our goals is to prepare tests as early as possible. Most importantly, however, this kind of graph does not scale to large software products. The graph is complicated enough with small Stutter and gets much worse with larger programs.

Figure 2.36. A FSM representing stutter, based on control flow graphs of the methods.

2. *Using the software structure*: A more experienced programmer may consider the overall flow of operations in the software. This might lead to something like the graph in Figure 2.37.

Although an improvement over the control flow graph, the kind of derivation shown in Figure 2.37 is very subjective. Different testers will draw different graphs, which introduces inconsistency in the testing. It also requires in-depth knowledge of the software, is not possible until at least the detailed design is ready, and is hard to scale to large programs.

3. *Modeling state variables*: A more mechanical method for deriving FSMs is to consider the values of the state variables in the program. These are usually defined early in design. The first step is to identify the state variables, then choose which ones are actually relevant to the FSM (for example, global and class variables). Class `Stutter` defines four state variables, `lastdelimit`, `curWord`, `prevWord`, and `delimits`. The variable `delimits` is defined at the class level for convenience, but should not

Figure 2.37. A FSM representing Stutter, based on the structure of the software.

really be considered part of the state. In fact, it is possible that it will not even be included in the design. However, lastdelimit, curWord, and prevWord are true state variables.

Theoretically, each combination of values for the three state variables defines a different state. In practice, however, this can result in an infinite number of states. For example, curWord and prevWord are strings and have an infinite number of values. Thus, it is common to identify values or ranges of values that should be represented as states. For class Stutter, the obvious approach is to consider the relationship between the two variables, yielding the following possible values:

```
curWord: undefined, word
prevWord: undefined, sameword, differentword
lastdelimit: true, false
```

where word is an arbitrary string. This combination of values leads to twelve possible states:

1. (undefined, undefined, true)
2. (undefined, undefined, false)
3. (undefined, sameword, true)
4. (undefined, sameword, false)
5. (undefined, differentword, true)
6. (undefined, differentword, false)
7. (word, undefined, true)
8. (word, undefined, false)
9. (word, sameword, true)
10. (word, sameword, false)

Figure 2.38. A FSM representing Stutter, based on modeling state variables.

11. (word, differentword, true)
12. (word, differentword, false)

Not every combination is possible. For example, curWord is given a value immediately and never becomes undefined thereafter. So the only possible state where curWord has the value undefined is state 1, which also happens to be the initial state. As soon as a character is read, curWord has value word and lastdelimit is set to false (state 8). When a delimiter is found, prevWord is either the same (sameword) or different (different word) from curWord (states 10 or 12). If the two words are the same, the next character read changes curWord, so the software transitions to (word, differentword, false), state 12. The complete state machine is derived in this way, by deciding all possible transitions from the states. The complete FSM for Stutter is shown in Figure 2.38. Note that it is impossible to get to state 7, so it is omitted from the FSM. Also, note that the program terminates at the end of a file, which is always just after a delimiter is found. So every state in which lastdelimit = true is a final state.

The mechanical process of this strategy is appealing because we can expect different testers to derive the same or similar FSMs. It is not yet possible at this time to completely automate this process because of the difficulty of determining transitions from the source and because the decision of which variables to model requires judgment. The software is not necessary for this diagram to be derived, but the design is needed. The FSMs that are derived by modeling state variables may not accurately reflect the software.

4. *Using the implicit or explicit specifications*: The last method for deriving FSMs relies on explicit requirements or formal specifications describing the software's behavior. A FSM for class Stutter based on this approach is shown in Figure 2.39.

Figure 2.39. A FSM representing Stutter, based on the specifications.

This FSM looks a lot like the FSM based on the code, and this is to be expected. At the same time, FSMs based on specifications are usually cleaner and easier to understand. If the software is designed well, this type of FSM should contain the same information that UML state charts contain.

EXERCISES

Section 2.5.

1. Use the class **Queue** in Figure 2.40 for questions (a)–(f) below. The queue is managed in the usual circular fashion.
   Suppose we build a FSM where states are defined by the representation variables of **Queue**. That is, a state is a 4-tuple defined by the values for [*elements*, *size*, *front*, *back*]. For example, the initial state has the value [[*null*, *null*], 0, 0, 0], and the state that results from pushing an object *obj* onto the queue in its initial state is [[*obj*, *null*], 1, 0, 1].
   (a) We do not actually care which specific objects are in the queue. Consequently, there are really just four useful values for the variable *elements*. Enumerate them.
   (b) How many states are there?
   (c) How many of these states are reachable?
   (d) Show the reachable states in a drawing.
   (e) Add edges for the *enqueue*() and *dequeue*() methods. (For this assignment, ignore the exceptional returns, although you should observe that when exceptional returns are taken, none of the instance variables are modified.)
   (f) Define a small test set that achieves edge coverage. Implement and execute this test set. You might find it helpful to write a method that shows the internal variables at each call.

```
public class Queue
{ // Overview:  a Queue is a mutable, bounded FIFO data structure
  // of fixed size  (size is 2, for this exercise).
  // A typical Queue is [], [o1], or [o1, o2], where neither o1 nor o2
  // are ever null.  Older elements are listed before newer ones.
private Object[] elements;
private int size, front, back;
private static final int capacity = 2;

public Queue ()
{
   elements = new Object [capacity];
   size  = 0; front = 0; back  = 0;
}

public void enqueue (Object o)
    throws NullPointerException, IllegalStateException
{  // Modifies: this
   // Effects:   If argument is null throw NullPointerException
   // else if this is full, throw IllegalStateException,
   // else make o the newest element of this
   if (o == null)
      throw new NullPointerException ("Queue.enqueue");
   else if (size == capacity)
      throw new IllegalStateException ("Queue.enqueue");
   else
   {
      size++;
      elements [back] = o;
      back = (back+1) % capacity;
   }
}

public Object dequeue () throws IllegalStateException
{  // Modifies: this
   // Effects:   If queue is empty, throw IllegalStateException,
   // else remove and return oldest element of this

   if (size == 0)
      throw new IllegalStateException ("Queue.dequeue");
   else
   {
      size--;
      Object o = elements [ (front % capacity) ];
      elements [front] = null;
      front = (front+1) % capacity;
      return o;
   }
}

public boolean isEmpty() { return (size == 0); }
public boolean isFull() { return (size == capacity); }

public String toString()
{
   String result = "[";
   for (int i = 0; i < size; i++)
   {
      result += elements[ (front + i) % capacity ] . toString();
      if (i < size -1) {
         result += ", ";
      }
   }
   result += "]";
   return result;
}
}
```

Figure 2.40. Class Queue for exercises.

2. For the following questions (a)–(c), consider the method FSM for a (simplified) programmable thermostat. Suppose the variables that define the state and the methods that transition between states are:

```
partOfDay : {Wake, Sleep}
temp      : {Low, High}

// Initially "Wake" at "Low" temperature

// Effects: Advance to next part of day
public void advance();

// Effects: Make current temp higher, if possible
public void up();

// Effects: Make current temp lower, if possible
public void down();
```

   (a) How many states are there?
   (b) Draw and label the states (with variable values) and transitions (with method names). Notice that all of the methods are total.
   (c) A test case is simply a sequence of method calls. Provide a test set that satisfies edge coverage on your graph.

2.6 GRAPH COVERAGE FOR USE CASES

UML use cases are widely used to clarify and express software requirements. They are meant to describe sequences of actions that software performs as a result of inputs from the users; that is, they help express the **workflow** of a computer application. Because use cases are developed early in software development, they can be help the tester start testing activities early.

Many books and papers can help the reader develop use cases. As with FSMs, it is not the purpose of this book to explain how to develop use cases, but how to use them to create useful tests. The technique for using graph coverage criteria to develop tests from use cases is expressed with a simple example.

Figure 2.41 shows three simple use cases for an automated teller machine (ATM). In use cases, *actors* are humans or other software systems that use the software being modeled. They are drawn as simple stick figures. In Figure 2.41, the actor is an ATM customer who has three potential use cases; Withdraw Funds, Get Balance, and Transfer Funds.

While Figure 2.41 is technically a graph, it is not a very useful graph for testing. About the best we could do as a tester is to use node coverage, which amounts to "try each use case once." However, use cases are usually elaborated, or "documented" with a more detailed textual description. The description describes the details of operation and includes *alternatives*, which model choices or conditions during execution. The Withdraw Funds use case from Figure 2.41 can be described as follows:

Figure 2.41. ATM actor and use cases.

Use Case Name: Withdraw Funds
Summary: Customer uses a valid card to withdraw funds from a valid bank account.
Actor: ATM Customer
Precondition: ATM is displaying the idle welcome message
Description:

1. Customer inserts an ATM Card into the ATM Card Reader.
2. If the system can recognize the card, it reads the card number.
3. System prompts the customer for a PIN.
4. Customer enters PIN.
5. System checks the expiration date and whether the card has been stolen or lost.
6. If card is valid, the system checks whether the PIN entered matches the card PIN.
7. If the PINs match, the system finds out what accounts the card can access.
8. System displays customer accounts and prompts the customer to choose a type of transaction. Three types of transactions are Withdraw Funds, Get Balance, and Transfer Funds. (The previous eight steps are part of all three use cases; the following steps are unique to the Withdraw Funds use case.)
9. Customer selects Withdraw Funds, selects account number, and enters the amount.
10. System checks that the account is valid, makes sure that the customer has enough funds in the account, makes sure that the daily limit has not been exceeded, and checks that the ATM has enough funds.
11. If all four checks are successful, the system dispenses the cash.
12. System prints a receipt with a transaction number, the transaction type, the amount withdrawn, and the new account balance.
13. System ejects card.
14. System displays the idle welcome message.

Alternatives:

- If the system cannot recognize the card, it is ejected and a welcome message is displayed.
- If the current date is past the card's expiration date, the card is confiscated and a welcome message is displayed.

- If the card has been reported lost or stolen, it is confiscated and a welcome message is displayed.
- If the customer entered PIN does not match the PIN for the card, the system prompts for a new PIN.
- If the customer enters an incorrect PIN three times, the card is confiscated and a welcome message is displayed.
- If the account number entered by the user is invalid, the system displays an error message, ejects the card, and a welcome message is displayed.
- If the request for withdrawal exceeds the maximum allowable daily withdrawal amount, the system displays an apology message, ejects the card, and a welcome message is displayed.
- If the request for withdrawal exceeds the amount of funds in the ATM, the system displays an apology message, ejects the card, and a welcome message is displayed.
- If the customer enters Cancel, the system cancels the transaction, ejects the card, and a welcome message is displayed.
- If the request for withdrawal exceeds the amount of funds in the account, the system displays an apology message, cancels the transaction, ejects the card, and a welcome message is displayed.

Postcondition: Funds have been withdrawn from the customer's account.

At this point, some testing students will be wondering why this discussion is included in a chapter on graph coverage. That is, there is little obvious relationship with graphs thus far. We want to reiterate the first phrase in Beizer's admonition: "testers find a graph, then cover it." In fact, there is a nice graph structure in the use case textual description, which may be up to the tester to express. This graph can be modeled as the transaction flow graphs in Beizer's Chapter 4, or can be drawn as a UML **Activity Diagram**.

An activity diagram shows the flow among activities. Activities can be used to model a variety of things, including state changes, returning values, and computations. We advocate using them to model use cases as graphs by considering activities as *user level steps*. Activity diagrams have two kinds of nodes, action states and sequential branches.[6]

We construct activity graphs as follows. The numeric items in the use case **Description** express steps that the actors undertake. These correspond to inputs to or outputs from the software and appear as **nodes** in the activity diagram as action states. The **Alternatives** in the use case represent decisions that the software or actors make and are represented as **nodes** in the activity diagram as sequential branches.

The activity diagram for the withdraw funds scenario is shown in Figure 2.42. Several things are **expected** but not **required** of activity diagrams constructed from use cases. First, they usually do not have many loops, and most loops they do contain are tightly bounded or determinate. For example, the graph in Figure 2.42 contains a three-iteration loop when the PIN is entered incorrectly. This means that complete path coverage is often feasible and sometimes reasonable. Second, it is very rare to see a complicated predicate that contains multiple clauses. This is because the use case is usually expressed in terms that the users can understand. This means that the logic coverage criteria in Chapter 3 are usually not useful. Third, there are no

Figure 2.42. Activity graph for ATM withdraw funds.

obvious data definition-use pairs. This means that data flow coverage criteria are not applicable.

The two criteria that are most obviously applicable to use case graphs are node coverage and edge coverage. Test case values are derived from interpreting the nodes and predicates as inputs to the software. One other criterion for use case graphs is based on the notion of "scenarios."

2.6.1 Use Case Scenarios

A use case scenario is an *instance* of, or a complete path through, a use case. A scenario should make some sense semantically to the users and is often derived when the use cases are constructed. If the use case graph is finite (as is usually the case), then it is possible to list all possible scenarios. However, domain knowledge can be used to reduce the number of scenarios that are useful or interesting from either a modeling or test case perspective. Note that *specified path coverage*, defined at the beginning of this chapter, is exactly what we want here. The set S for specified path coverage is simply the set of all scenarios.

If the tester or requirements writer chooses all possible paths as scenarios, then specified path coverage is equivalent to complete path coverage. The scenarios are chosen by people and they depend on domain knowledge. Thus it is **not** guaranteed that specified path coverage subsumes edge coverage or node coverage. That is, it is possible to choose a set of scenarios that do not include every edge. This would probably be a mistake, however. So, in practical terms, specified path coverage can be expected to cover all edges.

EXERCISES

Section 2.6.

1. Construct two separate use cases and use case scenarios for interactions with a bank automated teller machine. Do not try to capture all the functionality of the ATM into one graph; think about two different people using the ATM and what each one might do.

   Design test cases for your scenarios.

2.7 REPRESENTING GRAPHS ALGEBRAICALLY

While we typically think of graphs as circles and arrows, they can be represented in various nonpictorial ways. One useful way is an algebraic representation, which can be manipulated using standard algebraic operations and converted to regular expressions. These operations can then be used as a basis for testing the software and to answer various questions about the graphs.

The first requirement is that each edge have a unique label or name. The edge names can come from labels that are already associated with the edges, or can be added specifically for the algebraic representation. This book assumes the labels are unique lower case letters. The multiplicative operator in graph algebra is concatenation; if edge a is followed by edge b, their product is ab (the operator '*' is not written explicitly). The additive operator is selection; if either edge a or edge b can be taken, their sum is $a + b$. Concatenating edges together forms a path, so a sequence of edges is called a *path product*. A *path expression* contains path products and zero or more '+' operators. Thus, every path product is a path expression. Note that an edge label is a special case of a path product with no multiplication, and a path product is a special case of a path expression with no addition. Path expressions are sometimes represented by upper case letters, for example, $A = ab$.

Figure 2.43 shows three example graphs drawn from the double-diamond graph, the loop touring graph, and the Stutter example from previous sections. Figure 2.43(a) has exactly four paths, all of which are shown. Figure 2.43(b) and (c) include loops, so not all paths are shown. In graph algebra, loops are best represented using exponents.

If an edge, path product, or path expression can be repeated, then it is labeled with an exponent. Therefore, $a^2 = aa$, $a^3 = aaa$, and $a^* = aa \cdots a$, that is, an arbitrary number of repetitions. As a special case, an empty, or zero length path, can be represented by $a^0 = \lambda$. This makes λ the multiplicative identity, so $a\lambda = a$, or more generally, $A\lambda = A$.

Representing paths or partial paths with upper case letters makes for convenient manipulation. For example, we can take some partial path expressions from Figure 2.43(b) above:

$$\begin{aligned} A &= ab \\ B &= eg \\ C &= cd \\ AB &= abeg \\ C^3 &= cdcdcd \end{aligned}$$

(a) A double diamond graph with edge labels

(b) A graph with a loop

(c) The first FSM for stutter

Figure 2.43. Examples of path products.

$$
\begin{aligned}
AC^2B &= ab(cd)^2eg \\
&= abcdcdeg \\
D &= be + bcf
\end{aligned}
$$

Unlike standard algebra, path products are **not** commutative. That is, $AB \neq BA$. They are, however, associative, so $A(BC) = (AB)C = ABC$.

All paths in the graph in Figure 2.43(a) above can be represented by the expression: $abdfhj + abdgij + acefhj + acegij$. Paths that are summed can be considered to be independent or parallel paths. So path summation is both commutative and associative, that is, $A + B = B + A$, $(A + B) + C = A + (B + C) = A + B + C$.

With this basis, we can start applying standard algebraic laws. Both the distributive law and absorption rule can be applied.

$$
\begin{aligned}
A(B+C) &= AB + AC \text{ (distributive)} \\
(B+C)D &= BD + CD \text{ (distributive)} \\
A + A &= A \text{ (absorption rule)}
\end{aligned}
$$

We also have two more shorthand notations for repetition or loops. If a loop has to be taken at least once (for example, a repeat-until structure), then the '+' exponent is used. That is, $AA^* = A^+$. We can also put bounds on repetition if a loop has a definite bound (for example, a **for** loop). This is done with an underscore: $A^{\underline{3}} = A^0 + A^1 + A^2 + A^3$, or more generally, $A^{\underline{n}} = A^0 + A^1 + \cdots + A^n$. It is sometimes helpful to bound the number of iterations on both ends – that is, at least m and at most n iterations are possible. To do this, we introduce the notation $A^{\underline{m-n}} = A^m + A^{m+1} + \cdots + A^n$.

The absorption rule can be used to combine the exponent notations in several ways. This is used to simplify path expressions as follows:

$$\begin{aligned}
A^{\underline{n}} + A^{\underline{m}} &= A^{\underline{max\ (n,m)}} \\
A^{\underline{n}} A^{\underline{m}} &= A^{\underline{n+m}} \\
A^{\underline{n}} A^* &= A^* A^{\underline{n}} = A^* \\
A^{\underline{n}} A^+ &= A^+ A^{\underline{n}} = A^+ \\
A^* A^+ &= A^+ A^* = A^+
\end{aligned}$$

The multiplicative identity operator, λ, can also be used to simplify path expressions.

$$\begin{aligned}
\lambda + \lambda &= \lambda \\
\lambda A &= A\lambda = A \\
\lambda^n &= \lambda^{\underline{n}} = \lambda^* = \lambda^+ = \lambda \\
\lambda^+ + \lambda &= \lambda^* = \lambda
\end{aligned}$$

We also need an additive identity. We will use ϕ to represent the set of paths that contains no paths (not even the empty path λ). Mathematically, any path expression added to ϕ is just that path expression. The additive ϕ can be thought of as "blocking" the paths in the graph, therefore making a null path.

$$\begin{aligned}
A + \phi &= \phi + A = A \\
A\phi &= \phi A = \phi \\
\phi^* &= \lambda + \phi + \phi^2 + \cdots = \lambda
\end{aligned}$$

Figure 2.44 shows a small graph that has a null path. If we list all paths from node n_0 to n_3, we get the path expression $bc + a\phi = bc$.

A special case is the path expression $A + \lambda$. This situation is illustrated in Figure 2.45. The complete path expression is $(A + \lambda)B$, or $AB + \lambda B$, or $AB + B$. Thus, $A + \lambda$ cannot be reduced.

Figure 2.44. Null path that leads to additive identity ϕ.

Figure 2.45. A or lambda.

2.7.1 Reducing Graphs to Path Expressions

Now that we have the basic tools, we can see how to go about reducing arbitrary graphs to path expressions. The process given here is not a strict algorithm as it requires some thought and decisions, but is good enough to be used by human testers. As far as we know, this process has not been automated and implemented in a tool; however, it is a special case of the common technique of constructing regular expressions from deterministic FSMs. The process is illustrated on the graph shown in Figure 2.46.

Step 1: First we combine all sequential edges, multiplying the edge labels. More formally, for any node that has only one incoming edge and one outgoing edge, eliminate the node, combine the two edges, and multiply their path expressions. Applying this step to the graph in Figure 2.46 combines edges h and i, giving the graph shown in Figure 2.47.

Step 2: Next combine all parallel edges, adding the edge labels. More formally, for any pair of edges that have the same source and target nodes, combine the edges into one edge, and add their path expressions. The graph in Figure 2.47 contains one such pair of edges, b and c, so they are combined to yield $b + c$, giving the graph shown in Figure 2.48.

Step 3: Remove self-loops (from a node to itself) by creating a new "dummy" node with an incoming edge that uses the exponent operator '*', then merging the three edges with multiplication. More formally, for any node n_1 that has an edge to itself with label X, and incoming edge A and outgoing edge B, remove the edge with label X, and add a new node n'_1 and an edge with label X^*. Then combine the three edges A, X^*, and B into one edge AX^*B (eliminating nodes n_1 and n'_1). The graph in Figure 2.48 contains one self-loop on node n_3 with label e. The edge is first replaced with node n'_3 and an edge from n_3 to n'_3 with label e^* (as shown in Figure 2.49(a)), then the edges labeled d, e^* and f are combined, as shown in Figure 2.49(b).

Step 4: Now the tester starts choosing nodes to remove. Select a node that is not the initial or final node. Replace it by inserting edges from all predecessors to all successors, multiplying the path expressions from the incoming to the outgoing edges. Figure 2.50 illustrates this with a node that has two incoming and two outgoing edges.

Figure 2.46. Example graph to show reduction to path expressions.

Figure 2.47. After step 1 in path expression reduction.

Figure 2.48. After step 2 in path expression reduction.

Figure 2.49. After step 3 in path expression reduction.

Figure 2.50. Removing arbitrary nodes.

Figure 2.51. Eliminating node n_2.

Figure 2.52. Removing sequential edges.

Figure 2.53. Removing self-loop edges.

Node n_2 in Figure 2.49(b) has two incoming edges and one outgoing edge. Edges (n_1, n_2) and (n_2, n_4) become edge (n_1, n_4), with the two path expressions multiplied, and edges (n_4, n_2) and (n_2, n_4) become a self-loop (n_4, n_4), with the two path expressions multiplied. The resulting graph is shown in Figure 2.51.

Steps 1 through 4 are repeated until only one edge remains in the graph. Applying step 1 (combining sequential edges) again to the graph in Figure 2.51 yields the graph shown in Figure 2.52.

Applying step 2 (combining parallel edges) again is skipped because the graph in Figure 2.52 has no parallel edges. Applying step 3 (removing self-loops) again to the graph in Figure 2.52 removes the self-loop on node n_4, yielding the graph shown in Figure 2.53. The final graph (and regular expression) in our example is shown in Figure 2.54.

2.7.2 Applications of Path Expressions

Now that the mathematical preliminaries are out of the way, it is fair to ask what do we do with these path expressions? Path expressions are abstract, formal representations of graphs. As such, they can be manipulated to give us information about the graphs they represent. This section presents several applications of path expressions.

2.7.3 Deriving Test Inputs

The most direct way to use path expression representations of graphs is to define covering test cases. Each path, that is, each path product, defined by the path expression should be executed, with an appropriate limitation on loops. This is a form of *specified path coverage (SPC)*. If an unbounded exponent ('*') appears in the path expression, it can be replaced with one or more reasonably chosen constant values, then a complete list of paths can be written out. This technique will ensure (that is, subsume) node coverage and edge coverage on the graph.

The final path expression for the example in Figures 2.46 through 2.54 is $abde^* f(gde^* f)^* hi + acde^* f(gde^* f)^* hi$. This expression has two separate path products, and the exponents can be replaced (arbitrarily) by the constant 5. This results in the following two test requirements: $abde^5 f(gde^5 f)^5 hi$ and $acde^5 f(gde^5 f)^5 hi$.

Figure 2.54. Final graph with one path expression.

2.7.4 Counting Paths in a Flow Graph and Determining Max Path Length

It is sometimes useful to know the number of paths in a graph. This can be used as a simplistic complexity measure or as a rough estimation of the number of tests needed to cover the graph. The path expressions allow this computation with straightforward arithmetic, yielding a reasonable approximation for the maximum number of paths.

As discussed earlier, whenever a graph has a cycle, theoretically the graph has an infinite number of paths. However, some graphs have no cycles, and domain knowledge can be used to put appropriate bounds on the number of iterations. The bound may be a true maximum number of iterations, or it may represent a tester's assumption that executing the loop "N times" is enough.

The first step is to label each edge with an *edge weight*. For most edges, the edge weight is one. If the edge represents an expensive operation, such as a method call or external process, the edge weight should be the approximate weight of that operation (for example, the number of paths in the method). If the edge represents a cycle, mark it with the maximum number of iterations possible (the *cycle weight*). It is possible that this number is infinite, which means the maximum number of paths in the graph is infinite. It is important that not all edges in a cycle be labeled with the cycle weight. Only one edge per each cycle should be labeled. Sometimes, which edge to label is obvious, other times the tester must choose an appropriate edge, taking care not to omit a cycle or label a cycle twice. Consider graphs (b) and (c) in Figure 2.43. It should be clear that the cycle weight should be placed on edge d in graph (b). Cycle weights should also be placed on edges h, i, and j in graph (c), and on both edges f and g. Edge f will always occur on any path that includes edge g, so it is easy to forget one of those cycle weights; however, they represent separate cycles.

Sometimes we want to separately annotate a loop to indicate how many times it can be taken. The notation "(0–10)" means that the loop can be taken 0 to 10 times inclusive. Note that this notation is **not** the same as the edge weight.

Next compute the path expression for the graph and substitute the weights into the path expression. The operators are used as one might expect. If the path expression is $A + B$, the substitution is $W_A + W_B$. If the path expression is AB, the substitution is $W_A * W_B$. If the path expression is $A^{\underline{n}}$, the substitution is the summation $\sum_{i=0}^{n} W_A^i$. If the path expression is $A^{\underline{m-n}}$, the substitution is the summation $\sum_{i=m}^{n} W_A^i$.

Figure 2.55 shows a simple graph with edge labels and edge weights. As indicated on edge d, the loop can be taken 0 to 2 times inclusive, and the edge weight for d is one. The resulting path expression is $a(b + c)(d(b + c))^{\underline{0-2}}e$.

Figure 2.55. Graph example for computing maximum number of paths.

The maximum number of paths can then be calculated by substituting the appropriate value for each edge label into the path expression.

$$
\begin{aligned}
& 1 * (1+1) * (1 * (1+1))^{\underline{0-2}} * 1 \\
= & 1 * 2 * 2^{\underline{0-2}} * 1 \\
= & 2 * \textstyle\sum_{i=0}^{2} 2^i * 1 \\
= & 2 * (2^0 + 2^1 + 2^2) * 1 \\
= & 2 * (1 + 2 + 4) * 1 \\
= & 2 * 7 * 1 \\
= & 14
\end{aligned}
$$

The length of the longest path in the graph can also be found. If the path expression is $A + B$, the substitution is $max(W_A, W_B)$. If the path expression is AB, the substitution is $W_A + W_B$. If the path expression is $A^{\underline{n}}$, the substitution is $n * W_A$. So the length of the longest path in the graph in Figure 2.55 is $1 + max(1, 1) + 2 * (1 + max(1, 1)) + 1 = 7$.

It is important to remember that these analyses do not include a feasibility analysis. Some paths may be infeasible, so this should be interpreted as an upper, or conservative, bound on the number of paths.

2.7.5 Minimum Number of Paths to Reach All Edges

A related question is how many paths have to be traversed to reach all edges. The process is very similar to counting the maximum number of paths and uses the same edge weights, but the computation is slightly different.

Specifically, if the path expression is $A + B$, the substitution is $W_A + W_B$. However, if the path expression is AB, the substitution is $max(W_A, W_B)$. If the path expression is $A^{\underline{n}}$, the substitution requires some judgment from the tester and is either 1 or W_A. If it is reasonable to assume that all paths through the loop can be taken during one test case, the value should be 1. If not, however, the value should be the weight of the loop, W_A. The second assumption is more conservative and leads to a higher number.

Again consider the graph in Figure 2.55. Assume that if the edge d is taken, the same edge that preceded it must then be taken. That is, if b is taken, then d, the logic of the graph dictates that b must be taken again. This means that we must use the conservative estimate for the loop, yielding

$$
\begin{aligned}
& 1 * (2) * (1 * (2))^{\underline{2}} * 1 \\
= & 1 * (2) * (1 * 2) * 1 \\
= & max(1, 2, 1, 2, 1) \\
= & 2
\end{aligned}
$$

A visual inspection of the graph can confirm that all edges can be reached with two traversals of the graph.

2.7.6 Complementary Operations Analysis

The last application of path expressions is not a counting application, but an analysis that looks for anomalies that may lead to mistakes. It is based on the idea of "complementary operations." Two operations are *complementary* if their behaviors negate each other, or one must be done before the other. Examples include push

Figure 2.56. Graph example for complementary path analysis.

and pop in stacks, enqueue and dequeue in queues, get and dispose for memory, and open and close for files.

The process starts with the path expression for a graph, except instead of edge weights, each edge is marked with one of the following three labels:

1. C – Creator operation (push, enqueue, etc.)
2. D – Destructor operation (pop, dequeue, etc.)
3. 1 – Neither a creator nor a destructor

The path expression multiplicative and additive operators are replaced with the following two tables[7]:

*	C	D	1
C	C^2	1	C
D	DC	D^2	D
1	C	D	1

+	C	D	1
C	C	$C+D$	$C+1$
D	$D+C$	D	$D+1$
1	$1+C$	$1+D$	1

Note the differences from the usual algebra defined on integers. $C * D$ reduces to 1, $C + C$ reduces to C, and $D + D$ reduces to D.

Consider the graph in Figure 2.56. Edges are marked with C, D or 1, and its initial path expression is $C(C+1)C(C+D)1(D(C+D)1)^n1$. The algebraic rules are used to rewrite this as $(CCCC + CCCD + CCC + CCD)(DC + DD)^n$. The two tables above can be used to further reduce the path expression to $(CCCC + CC + CCC + C)(DC + DD)^n$.

The first question to ask of this path expression is "is it possible to have more destruct operations than creates?" The answer is yes, and some expressions are

$CCCD(DD)^n, n > 1$
$CCD(DD)^n, n > 0$
$CCC(DDDCDD)$

Another question is "is it possible to have more create operations than destructs?" Again, the answer is yes, and some expressions are:

$CCCC$
$CCD(DC)^n, \forall n$

Each yes answer represents a specification for a test that is likely to cause anomalous behavior.

EXERCISES

Section 2.7.

1. Derive and simplify the path expressions for the three graphs in Figure 2.43.
2. Derive and simplify the path expression for the flow graph in Figure 2.12. Assign reasonable cycle weights and compute the maximum number of paths in the graph and the minimum number of paths to reach all edges.

3. The graph in Figure 2.10 was used as an example for prime test paths. Add appropriate edge labels to the graph, then derive and simplify the path expressions. Next add edge weights of 1 for non-cycle edges and 5 for cycle edges. Then compute the maximum number of paths in the graph and the minimum number of paths to reach all edges. This graph has 25 prime paths. Briefly discuss the number of prime paths with the maximum number of paths and consider the effect of varying the cycle weight on the maximum number of paths.
4. Section 2.5 presented four different versions of a FSM for Stutter. Derive and simplify the path expressions for each of the four variations, then compute the maximum number of paths and the minimum number of paths to reach all edges in each. Discuss how the different numbers affect testing.
5. Perform complementary operations analysis on the graph in Figure 2.32. Assume complementary operators of **open** and **close**.
6. Derive and simplify the path expressions for the activity graph in Figure 2.42. The only loop in that graph has a bound of 3 tries. Use that to compute the maximum number of paths and the minimum number of paths to reach all edges. Discuss the relationship between the scenarios for that graph and the terms in the path expression.
7. Answer questions (a)–(c) for the graph defined by the following sets:
   - $N = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
   - $N_0 = \{1\}$
   - $N_f = \{10\}$
   - $E = \{(1, 2, a), (2, 3, b), (2, 4, c), (3, 5, d), (4, 5, e), (5, 6, f), (5, 7, g), (6, 6, h(1-4)), (6, 10, i), (7, 8, j), (8, 8, k(0-3)), (8, 9, l), (9, 7, m(2-5)), (9, 10, n)\}$

   (a) Draw the graph.
   (b) What is the maximum number of paths through the graph?
   (c) What is the approximate minimum number of paths through the graph?

2.8 BIBLIOGRAPHIC NOTES

During the research for this book, one thing that became abundantly clear is that this field has had a significant amount of parallel discovery of the same techniques by people working independently. Some individuals have discovered various aspects of the same technique, which was subsequently polished into very pretty test criteria. Others have invented the same techniques, but based them on different types of graphs or used different names. Thus, ascribing credit for software testing criteria is a perilous task. We do our best, but claim only that the bibliographic notes in this book are starting points for further study in the literature.

The research into covering graphs seems to have started with generating tests from finite state machines (FSMs), which has a long and rich history. Some of the earliest papers were in the 1970s [77, 164, 170, 232, 290]. The primary focus of most of these papers was on using FSMs to generate tests for telecommunication systems that were defined with standard finite automata, although much of the work pertained to general graphs. The control flow graph seems to have been invented (or should it be termed "discovered"?) by Legard in 1975 [204]. In papers published

in 1975, Huang [170] suggested covering each edge in the FSM, and Howden [164] suggested covering complete trips through the FSM, but without looping. In 1976, McCabe [232] suggested the same idea on control flow graphs as the primary application of his cyclomatic complexity metric. In 1976, Pimont and Rault [290] suggested covering pairs of edges, using the term "switch cover." In 1978, Chow [77] suggested generating a spanning tree from the FSM and then basing test sequences on paths through this tree. In 1991, Fujiwara et al. [130] extended Pimont and Rault's pairs of edges to arbitrary lengths, and used the term "n-switch" to refer to a sequence of edges. He also attributed "1-switch," or switch cover, to Chow and called it the "W-method," an inaccuracy that has been repeated in numerous papers. The idea of covering pairs of edges was rediscovered in the 1990s. The British Computer Society Standard for Software Component Testing called it *two-trip* [317] and Offutt et al. [272], called it *transition-pair*.

Other test generation methods based on FSMs include tour [251], the distinguished sequence method [137], and unique input-output method [307]. Their objectives are to detect output errors based on state transitions driven by inputs. FSM-based test generation has been used to test a variety of applications including lexical analyzers, real-time process control software, protocols, data processing, and telephony. One early realization when developing this book is that the criteria for covering FSMs are not substantially different from criteria for other graphs.

This book has introduced the explicit inclusion of node coverage requirements in edge coverage requirements (the "up to" clause). This inclusion is not necessary for typical control flow graphs, where, indeed, subsumption of node coverage by edge coverage is often presented as a basic theorem, but it may be required for graphs derived from other artifacts.

Several later papers focused on automatic test data generation to cover structural elements in the program [39, 41, 80, 101, 117, 166, 190, 191, 267, 295]. Much of this work was based on the analysis techniques of symbolic evaluation [62, 83, 93, 101, 116, 164], and slicing [328, 339]. Some of these ideas are discussed in Chapter 6.

The problem of handling loops has plagued graph-based criteria from the beginning. It seems obvious that we want to cover paths, but loops create infinite numbers of paths. In Howden's 1975 paper [164], he specifically addressed loops by covering complete paths "without looping," and Chow's 1978 suggestion to use spanning trees was an explicit attempt to avoid having to execute loops [77]. Binder's book [33] used the technique from Chow's paper, but changed the name to *round trip*, which is the name used in this book.

Another early suggestion was based on testing loop free programs [66], which is certainly interesting from a theoretical view, but not particularly practical.

White and Wiszniewski [348] suggested limiting the number of loops that need to be executed based on specific patterns. Weyuker, Weiss, and Hamlet tried to choose specific loops to test based on data definitions and uses [345].

The notion of *subpath sets* was developed by Offutt et al. [178, 265] to support inter-class path testing and is essentially equivalent to tours with detours as presented here. The ideas of touring, sidetrips and detours were introduced by Ammann, Offutt and Huang [17].

The earliest reference we have found on data flow testing was a technical report in 1974 by Osterweil and Fosdick [282]. This technical report was followed by a 1976

paper in *ACM Computing Surveys* [122], *along with an almost simultaneous publication by Herman in the Australian Computer Journal* [158]. The seminal data flow analysis procedure (without reference to testing) was due to Allen and Cocke [13].

Other fundamental and theoretical references are by Laski and Korel in 1983 [201], who suggested executing paths from definitions to uses, Rapps and Weyuker in 1985 [297], who defined criteria and introduced terms such as All-Defs and All-Uses, and Frankl and Weyuker in 1988 [128]. These papers refined and clarified the idea of data flow testing, and are the basis of the presentation in this text. Stated in the language in this text, [128] requires direct tours for the All-du-Paths Coverage, but allows sidetrips for All-Defs coverage and All-Uses coverage. This text allows sidetrips (or not) for all of the data flow criteria. The pattern matching example used in this text has been employed in the literature for decades; as far as we know, Frankl and Weyuker [128] were the first to use the example for illustrating data flow coverage.

Forman also suggested a way to detect data flow anomalies without running the program [121].

Some detailed problems with data flow testing have been recurring. These include the application of data flow when paths between definitions and uses cannot be executed [127], and handling pointers and arrays [267, 345].

The method of defining data flow criteria in terms of sets of du-paths is original to this book, as is the explicit suggestion for best-effort eouring.

Many papers present empirical studies of various aspects of data flow testing. One of the earliest was by Clarke, Podgurski, Richardson, and Zeil, who compared some of the different criteria [82]. Comparisons with mutation testing (introduced in Chapter 5) started with Mathur in 1991 [228], which was followed by Mathur and Wong [230], Wong and Mathur [357], Offutt, Pan, Tewary, and Zhang [274], and Frankl, Weiss, and Hu [125]. Comparisons of data flow with other test criteria have been published by Frankl and Weiss [124], Hutchins, Foster, Goradia, and Ostrand [172], and Frankl and Deng [123].

A number of tools have also been built by researchers to support data flow testing. Most worked by taking a program and tests as inputs, and deciding whether one or more data flow criteria have been satisfied (a *recognizer*). Frankl, Weiss, and Weyuker built ASSET in the mid 1980s [126], Girgis and Woodward built a tool to implement both data flow and mutation testing in the mid 1980s [134], and Laski built STAD in the late 1980s [200]. Researchers at Bellcore developed the ATAC data flow tool for C programs in the early 1990s [161, 162], and the first tool that included a test data generator for data flow criteria was built by Offutt, Jin, and Pan in the late 1990s [267].

Coupling was first discussed as a design metric by Constantine and Yourdon [88], and its use for testing was introduced implicitly by Harrold, Soffa, and Rothermel [152, 154] and explicitly by Jin and Offutt [178], who introduced the use of *first-uses* and *last-defs*.

Kim, Hong, Cho, Bae, and Cha used a graph-based approach to generate tests from UML state diagrams [186].

The USA's Federal Aviation Authority (FAA) has recognized the increased importance of modularity and integration testing by imposing requirements on structural coverage analysis of software that "the analysis should confirm the data

coupling and control coupling between the code components" [305], p. 33, section 6.4.4.2.

Data flow testing has also been applied to integration testing by Harrold and Soffa [154], Harrold and Rothermel [152], and Jin and Offutt [178]. This work focused on class-level integration issues, but did not address inheritance or polymorphism. Data flow testing has been applied to inheritance and polymorphism in object-oriented software by Alexander and Offutt [11, 10, 12], and Buy, Orso, and Pezze [60, 281]. Gallagher and Offutt modeled classes as interacting state machines, and tested concurrency and communication issues among them [132].

SCR was first discussed by Henninger [157], and its use in model checking and testing was introduced by Atlee [20].

Constructing tests from UML diagrams is a more recent development, though relatively straightforward. It was first suggested by Abdurazik and Offutt [2, 264], and soon followed by Briand and Labiche [45].

The mechanisms for turning finite automata into regular expressions are standard fare in CS theory classes. As far as we know, Beizer [29] was the first to note the utility of these transformations in the testing context.

NOTES

1 By way of example, typical control flow graphs have very few, if any, syntactically unreachable nodes, but call graphs, especially for object-oriented programs, often do.

2 Our mathematician readers might notice that this definition is constructive in that it defines what is in the set TR, but does not actually bound the set. It is certainly our intention that TR contains no other elements.

3 The reader might wonder why `NOTFOUND` fails to appear in the set *use*(2). The reason, as explained in Section 2.3.2 is that the use is *local*.

4 The reader is cautioned that despite the names of the criteria, All-Defs and All-Uses are not complementary criteria with respect to how they tread definitions and uses. Specifically, one does *not* arrive at All-Uses by replacing the notion of "def" with that of "use" in All-Defs. The reader might find it helpful to note that while All-Defs focuses on definitions, All-Uses focuses on def-use *pairs*. While one could argue that the naming convention is misleading, and that a name such as "All-Pairs" might be preferable to All-Uses, the authors elected to stick with the standard usage in the dataflow literature.

5 This is a bit of an overstatement, and, as usual, the culprit is infeasibility. Specifically, consider a du-path with respect to variable x that can only be toured with a sidetrip. Further, suppose that there are two possible sidetrips, one of which is def-clear with respect to x, and one of which is not. The relevant test path from the All-du-Paths test set necessarily tours the former sidetrip, where as the corresponding test path from the prime path test set is free to tour the latter side trip. Our opinion is that in most situations it is reasonable for the test engineer to ignore this special case and simply proceed with prime path coverage.

6 As in previous chapters, we explicitly leave out concurrency, so concurrent forks and joins are not considered.

7 Mathematicians who have studied abstract algebra will recognize that these tables define another algebra.

3

Logic Coverage

This chapter introduces test criteria based on logical expressions. While logic coverage criteria have been known for a long time, their use has been steadily growing in recent years. One cause for their use in practice has been their incorporation in standards such as those accepted by the US Federal Aviation Administration (FAA) for safety critical avionics software in commercial aircraft. As in Chapter 2, we start with a sound theoretical foundation for logic predicates and clauses with the goal of making the subsequent testing criteria simpler. As before, we take a generic view of the structures and criteria, then discuss how logic expressions can be derived from various software artifacts, including code, specifications, and finite state machines.

Readers who are already familiar with some of the common criteria may have difficulty recognizing them at first. This is because we introduce a generic collection of test criteria, and thus choose names that best help articulate all of the criteria. That is, we are abstracting a number of existing criteria that are closely related, yet use conflicting terminology.

3.1 OVERVIEW: LOGIC PREDICATES AND CLAUSES

We formalize logical expressions in a common mathematical way. A *predicate* is an expression that evaluates to a boolean value, and is our topmost structure. A simple example is: $((a > b) \vee C) \wedge p(x)$. Predicates may contain boolean variables, non-boolean variables that are compared with the comparator operators $\{>, <, =, \geq, \leq, \neq\}$, and function calls. The internal structure is created by the logical operators:

- $\neg$ – the *negation* operator
- $\wedge$ – the *and* operator
- $\vee$ – the *or* operator
- $\rightarrow$ – the *implication* operator
- $\oplus$ – the *exclusive or* operator
- $\leftrightarrow$ – the *equivalence* operator

Some of these operators ($\oplus$, $\rightarrow$, $\leftrightarrow$) may seem unusual for readers with a bias toward source code, but they turn out to be common in some specification languages and very handy in our purposes. Short circuit versions of the *and* and *or* operators are also sometimes useful and will be addressed when necessary. We adopt a typical precedence, which, from highest to lowest, matches the order listed above. When the order might not be obvious, we use parentheses for clarity.

A *clause* is a predicate that does not contain any of the logical operators. For example, the predicate $(a = b) \vee C \wedge p(x)$ contains three clauses: a relational expression $(a = b)$, a boolean variable C and the function call $p(x)$. Because they may contain a structure of their own, relational expressions require special treatment.

A predicate may be written in a variety of logically equivalent ways. For example, the predicate $((a = b) \vee C) \wedge ((a = b) \vee p(x))$ is logically equivalent to the predicate given in the previous paragraph, but $((a = b) \wedge p(x)) \vee (C \wedge p(x))$ is not. The usual rules of boolean algebra (not reviewed here) may be used to convert boolean expressions into equivalent forms.

Logical expressions come from a variety of sources. The most familiar to most readers will probably be source code of a program. For example, the following if statement:

```
if ((a > b) || C) && (x < y)
  o.m();
else
  o.n();
```

will yield the expression $((a > b) \vee C) \wedge (x < y)$. Other sources of logical expressions include transitions in finite state machines. A transition such as: `button2 = true` (when `gear = park`) will yield the expression $gear = park \wedge button2 = true$. Similarly, a precondition in a specification such as "`pre: stack Not full AND object reference parameter not null`" will result in a logical expression such as $\neg\, stackFull() \wedge newObj \neq null$.

In the material prior to Section 3.6 we treat logical expressions according to their semantic meanings, not their syntax. As a consequence, a given logical expression yields the same test requirements for a given coverage criterion no matter which form of the logic expression is used.

EXERCISES

Section 3.1.

1. List all the clauses for the predicate below:
   $((f <= g) \wedge (X > 0)) \vee (M \wedge (e < d + c))$
2. Write the predicate (only the predicate) to represent the requirement: "List all the wireless mice that either retail for more than \$100 or for which the store has more than 20 items. Also list non-wireless mice that retail for more than \$50."

3.2 LOGIC EXPRESSION COVERAGE CRITERIA

Clauses and predicates are used to introduce a variety of coverage criteria. Let P be a set of predicates and C be a set of clauses in the predicates in P. For each predicate $p \in P$, let C_p be the clauses in p, that is, $C_p = \{c | c \in p\}$. C is the union of the clauses in each predicate in P, that is, $C = \bigcup_{p \in P} C_p$.

> CRITERION **3.12 Predicate Coverage (PC):** *For each $p \in P$, TR contains two requirements: p evaluates to true, and p evaluates to false.*

The graph version of predicate coverage was introduced in Chapter 2 as edge coverage; this is where the graph coverage criteria overlap the logic expression coverage criteria. For control flow graphs where P is the set of predicates associated with branches, predicate coverage and edge coverage are the same. For the predicate given above, $((a > b) \vee C) \wedge p(x)$, two tests that satisfy predicate coverage are $(a = 5, b = 4, C = true, p(x) = true)$ and $(a = 5, b = 6, C = false, p(x) = false)$.

An obvious failing of this criterion is that the individual clauses are not always exercised. Predicate coverage for the above clause could also be satisfied with the two tests $(a = 5, b = 4, C = true, p(x) = true)$ and $(a = 5, b = 4, C = true, p(x) = false)$, in which the first two clauses never have the value *false*! To rectify this problem, we move to the clause level.

> CRITERION **3.13 Clause Coverage (CC):** *For each $c \in C$, TR contains two requirements: c evaluates to true, and c evaluates to false.*

Our predicate $((a > b) \vee C) \wedge p(x)$ requires different values to satisfy CC. Clause coverage requires that $(a > b) = true$ and *false*, $C = true$ and *false*, and $p(x) = true$ and *false*. These requirements can be satisfied with two tests: $((a = 5, b = 4), (C = true), p(x) = true)$ and $((a = 5,\ b = 6), (C = false), p(x) = false)$.

Clause coverage does not subsume predicate coverage, and predicate coverage does not subsume clause coverage, as we show with the predicate $p = a \vee b$. The clauses C are $\{a, b\}$. The four test inputs that enumerate the combinations of logical values for the clauses:

	a	b	$a \vee b$
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

Consider two test sets, each with a pair of test inputs. Test set $T_{23} = \{2, 3\}$ satisfies clause coverage, but not predicate coverage, because p is never false. Conversely, test set $T_{24} = \{2, 4\}$ satisfies predicate coverage, but not clause coverage, because b is never true. These two test sets demonstrate that neither predicate coverage nor clause coverage subsumes the other.

From the testing perspective, we would certainly like a coverage criterion that tests individual clauses and that also tests the predicate. The most direct approach to rectify this problem is to try all combinations of clauses:

CRITERION **3.14 Combinatorial Coverage (CoC):** *For each $p \in P$, TR has test requirements for the clauses in C_p to evaluate to each possible combination of truth values.*

Combinatorial coverage has also been called multiple condition coverage. For the predicate $(a \vee b) \wedge c$, the complete truth table contains eight elements:

	a	b	c	$(a \vee b) \wedge c$
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

A predicate p with n independent clauses has 2^n possible assignments of truth values. Thus combinatorial coverage is unwieldy at best, and impractical for predicates with more than a few clauses. What we need are criteria that capture the effect of each clause, but do so in a reasonable number of tests. These observations lead, after some thought,[1] to a powerful collection of test criteria that are based on the notion of making individual clauses "active" as defined in the next subsection. Specifically, we check to see that if we vary a clause in a situation where the clause should affect the predicate, then, in fact, the clause does affect the predicate. Later we turn to the complementary problem of checking to see that if we vary a clause in a situation where it should *not* affect the predicate, then it, in fact, does not affect the predicate.

3.2.1 Active Clause Coverage

The lack of subsumption between clause and predicate coverage is unfortunate, but clause and predicate coverage have deeper problems. Specifically, when we introduce tests at the clause level, we want also to have an effect on the predicate. The key notion is that of *determination*, the conditions under which a clause influences the outcome of a predicate. Although the formal definition is a bit messy, the basic idea is very simple: if you flip the clause, and the predicate changes value, then the clause determines the predicate. To distinguish the clause in which we are interested from the remaining clauses, we adopt the following convention. The *major* clause, c_i, is the clause on which we are focusing. All of the other clauses c_j, $j \neq i$, are *minor* clauses. Typically, to satisfy a given criterion, each clause is treated in turn as a major clause. Formally,

Definition 3.42 Determination: Given a major clause c_i in predicate p, we say that c_i *determines* p if the minor clauses $c_j \in p$, $j \neq i$ have values so that changing the truth value of c_i changes the truth value of p.

Note that this definition explicitly does **not** require that $c_i = p$. This issue has been left ambiguous by previous definitions, some of which require the predicate and the major clause to have the same value. This interpretation is not practical. When the negation operator is used, for example, if the predicate is $p = \neg a$, it becomes impossible for the major clause and the predicate to have the same value.

Consider the example above, where $p = a \vee b$. If b is false, then clause a determines p, because then the value of p is exactly the value of a. However if b is true, then a does not determine p, since p is true regardless of the value of a.

From the testing perspective, we would like to test each clause under circumstances where the clause determines the predicate. Consider this as putting different members of a team in charge of the team. We do not know if they can be effective leaders until they try. Consider again the predicate $p = a \vee b$. If we do not vary b under circumstances where b determines p, then we have no evidence that b is used correctly. For example, test set $T_{14} = \{TT, FF\}$, which satisfies both clause and predicate coverage, tests neither a nor b effectively.

In terms of criteria, we develop the notion of active clause coverage in a general way first with the definition below and then refine out the ambiguities in the definition to arrive at the resulting formal coverage criteria.

Definition 3.43 Active Clause Coverage (ACC): For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses c_j, $j \neq i$ so that c_i determines p. TR has two requirements for each c_i: c_i evaluates to true and c_i evaluates to false.

For example, for $p = a \vee b$, we end up with a total of four requirements in TR, two for clause a and two for clause b. For clause a, a determines p if and only if b is false. So we have the two test requirements $\{(a = true, b = false), (a = false, b = false)\}$. For clause b, b determines p if and only if a is false. So we have the two test requirements $\{(a = false, b = true), (a = false, b = false)\}$. This is summarized in the partial truth table below (the values for the major clauses are in bold face).

	a	b
$c_i = a$	**T**	f
	F	f
$c_i = b$	f	**T**
	f	**F**

Two of these requirements are identical, so we end up with three distinct test requirements for active clause coverage for the predicate $a \vee b$, namely, $\{(a = true, b = false), (a = false, b = true), (a = false, b = false)\}$. Such overlap always happens, and it turns out that for a predicate with n clauses, $n + 1$ distinct test requirements, rather than the $2n$ one might expect, are sufficient to satisfy active clause coverage.

ACC is almost identical to the way early papers described another technique called MCDC. It turns out that this criterion has some ambiguity, which has led to a fair amount of confusion about how to interpret MCDC over the years. The most important question is whether the minor clauses c_j need to have the same values when the major clause c_i is true as when c_i is false. Resolving this ambiguity leads to three distinct and interesting flavors of ACC. For a simple predicate such as $p = a \vee b$, the three flavors turn out to be identical, but differences appear for

more complex predicates. The most general flavor allows the minor clauses to have different values.

> CRITERION **3.15 General Active Clause Coverage (GACC):** *For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses c_j, $j \neq i$ so that c_i determines p. TR has two requirements for each c_i: c_i evaluates to true and c_i evaluates to false. The values chosen for the minor clauses c_j do* not *need to be the same when c_i is true as when c_i is false.*

Unfortunately, it turns out that GACC does not subsume predicate coverage, as the following example shows.

Consider the predicate $p = a \leftrightarrow b$. Clause a determines p for any assignment of truth values to b. So, when a is true, we choose b to be true as well, and when a is false, we choose b to be false as well. We make the same selections for clause b. We end up with only two test inputs: $\{TT, FF\}$. p evaluates to *true* for both of these cases, so predicate coverage is *not* achieved.

Many testing researchers have a strong feeling that ACC should subsume PC, thus the second flavor of ACC requires that p evaluates to true for one assignment of values to the major clause c_i, and false for the other. Note that c_i and p do not have to have the same values, as discussed with the definition for determination.

> CRITERION **3.16 Correlated Active Clause Coverage (CACC):** *For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses c_j, $j \neq i$ so that c_i determines p. TR has two requirements for each c_i: c_i evaluates to true and c_i evaluates to false. The values chosen for the minor clauses c_j must cause p to be true for one value of the major clause c_i and false for the other.*

So for the predicate $p = a \leftrightarrow b$ above, CACC can be satisfied with respect to clause a with the test set $\{TT, FT\}$ and with respect to clause b with the test set $\{TT, TF\}$. Merging these yields the CACC test set $\{TT, TF, FT\}$.

Consider the example $p = a \wedge (b \vee c)$. For a to determine the value of p, the expression $b \vee c$ must be true. This can be achieved in three ways: b true and c false, b false and c true, and both b and c true. So, it would be possible to satisfy CACC with respect to clause a with the two test inputs: $\{TTF, FFT\}$. Other choices are possible with respect to a. The following truth table helps enumerate them. The row numbers are taken from the complete truth table for the predicate given previously. Specifically, CACC can be satisfied for a by choosing one test requirement from rows 1, 2, and 3, and the second from rows 5, 6, and 7. Of course, nine possible ways exist to do this.

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F

The final flavor forces the c_j to be identical for both assignments of truth values to c_i.

> CRITERION **3.17 Restricted Active Clause Coverage (RACC):** *For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses c_j, $j \neq i$ so that c_i determines p. TR has two requirements for each c_i: c_i evaluates to true and c_i evaluates to false. The values chosen for the minor clauses c_j must be the same when c_i is true as when c_i is false.*

For the example $p = a \wedge (b \vee c)$, only three of the nine sets of test requirements that satisfy CACC with respect to clause a will satisfy RACC with respect to clause a. In terms of the previously given complete truth table, row 2 can be paired with row 6, row 3 with row 7, or row 1 with row 5. Thus, instead of the nine ways to satisfy CACC, only three can satisfy RACC.

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
5	F	T	T	F
2	T	T	F	T
6	F	T	F	F
3	T	F	T	T
7	F	F	T	F

CACC versus RACC

Examples of satisfying a predicate for each of these three criteria are given later. One point that may not be immediately obvious is how CACC and RACC differ in practice.

It turns out that some logical expressions can be completely satisfied under CACC, but have infeasible test requirements under RACC. These expressions are a little subtle and only exist if dependency relationships exist among the clauses, that is, some combinations of values for the clauses are prohibited. Since this often happens in real programs, because program variables frequently depend upon one another, it is useful to consider such an example.

Consider a system with a valve that might be either open or closed, and several modes, two of which are "Operational" and "Standby." Assume the following two constraints:

1. The valve must be open in "Operational" and closed in all other modes.
2. The mode cannot be both "Operational" and "Standby" at the same time.

This leads to the following clause definitions:

$a =$ "*The valve is closed*"
$b =$ "*The system status is Operational*"
$c =$ "*The system status is Standby*"

Suppose that a certain action can be taken only if the valve is closed and the system status is either in *Operational* or *Standby*. That is,

$$\begin{aligned} p &= \textit{valve is closed AND (system status is Operational OR} \\ &\quad \textit{system status is Standby)} \\ &= a \wedge (b \vee c) \end{aligned}$$

This is exactly the predicate that was analyzed above. The constraints above can be formalized as

1 $\neg a \leftrightarrow b$
2 $\neg(b \wedge c)$

These constraints limit the feasible values in the truth table. As a reminder, the complete truth table for this predicate is

	a	b	c	$a \wedge (b \vee c))$	
1	T	T	T	T	violates constraints 1 & 2
2	T	T	F	T	violates constraint 1
3	T	F	T	T	
4	T	F	F	F	
5	F	T	T	F	violates constraint 2
6	F	T	F	F	
7	F	F	T	F	violates constraint 1
8	F	F	F	F	violates constraint 1

Recall that for a to determine the value of P, either b or c or both must be true. Constraint 1 rules out the rows where a and b have the same values, that is, rows 1, 2, 7, and 8. Constraint 2 rules out the rows where b and c are both true, that is, rows 1 and 5. Thus, the only feasible rows are 3, 4, and 6. Recall that CACC can be satisfied by choosing one from rows 1, 2, or 3 and one from rows 5, 6, or 7. But RACC requires one of the pairs 2 and 6, 3, and 7, or 1 and 5. Thus, RACC is infeasible for a in this predicate.

3.2.2 Inactive Clause Coverage

The Active Clause Coverage criteria focus on making sure the major clauses *do* affect their predicates. A complementary criterion to ACC ensures that changing a major clause that should *not* affect the predicate does not, in fact, affect the predicate.

Definition 3.44 Inactive Clause Coverage (ICC): For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses c_j, $j \neq i$ so that c_i does *not* determine p. TR has four requirements for c_i under these circumstances: (1) c_i evaluates to true with p true, (2) c_i evaluates to false with p true, (3) c_i evaluates to true with p false, and (4) c_i evaluates to false with p false.

Although inactive clause coverage (ICC) has some of the same ambiguity as ACC does, only two distinct flavors can be defined, namely *general inactive clause*

coverage (GICC) and *restricted inactive clause coverage (RICC)*. The notion of correlation is not relevant for Inactive Clause Coverage because c_i cannot correlate with p since c_i does not determine p. Also, predicate coverage is guaranteed, subject to feasibility, in all flavors due to the structure of the definition.

The following example illustrates the value of the inactive clause coverage criteria. Suppose you are testing the control software for a shutdown system in a reactor, and the specification states that the status of a particular valve (open vs. closed) is relevant to the `reset` operation in `Normal` mode, but not in `Override` mode. That is, the `reset` should perform identically in `Override` mode when the valve is open and when the valve is closed. The sceptical test engineer will want to test `reset` in `Override` mode for both positions of the valve, since a reasonable implementation mistake would be to take account the setting of the valve in all modes.

The formal versions of GICC and RICC are as follows.

CRITERION **3.18 General Inactive Clause Coverage (GICC):** *For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses c_j, $j \neq i$ so that c_i does not determine p. TR has four requirements for c_i under these circumstances: (1) c_i evaluates to true with p true, (2) c_i evaluates to false with p true, (3) c_i evaluates to true with p false, and (4) c_i evaluates to false with p false. The values chosen for the minor clauses c_j may vary amongst the four cases.*

CRITERION **3.19 Restricted Inactive Clause Coverage (RICC):** *For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses c_j, $j \neq i$ so that c_i does* not *determine p. TR has four requirements for c_i under these circumstances: (1) c_i evaluates to true with p true, (2) c_i evaluates to false with p true, (3) c_i evaluates to true with p false, and (4) c_i evaluates to false with p false. The values chosen for the minor clauses c_j must be the same in cases (1) and (2), and the values chosen for the minor clauses c_j must also be the same in cases (3) and (4).*

3.2.3 Infeasibility and Subsumption

A variety of technical issues complicate the Active Clause Coverage criteria. As with many criteria, the most important is the issue of infeasibility. Infeasibility is often a problem because clauses are sometimes related to one another. That is, choosing the truth value for one clause may affect the truth value for another clause. Consider, for example, a common loop structure, which assumes short circuit semantics:

```
while (i < n && a[i] != 0) {do something to a[i]}
```

The idea here is to avoid evaluating `a[i]` if `i` is out of range, and short circuit evaluation is not only assumed, but depended on. Clearly, it is not going to be possible to develop a test case where `i < n` is false and `a[i] != 0` is true.

In principle, the issue of infeasibility for clause and predicate criteria is no different from that for graph criteria. In both cases, the solution is to satisfy test requirements that are feasible, and then decide how to treat infeasible test requirements.

Figure 3.1. Subsumption relations among logic coverage criteria.

The simplest solution is to simply ignore infeasible requirements, which usually does not affect the quality of the tests.

However, a better solution for some infeasible test requirements is to consider the counterparts of the requirements in a subsumed coverage criterion. For example, if RACC coverage with respect to clause a in predicate p is infeasible (due to additional constraints between the clauses), but CACC coverage is feasible, then it makes sense to replace the infeasible RACC test requirements with the feasible CACC test requirements. This approach is similar to that of best-effort touring developed in the graph coverage chapter.

Figure 3.1 shows the subsumption relationships among the logic expression criteria. Note that the ICC criteria do not subsume any of the ACC criteria, and vice versa. The diagram assumes that infeasible test requirements are treated on a best effort basis, as explained above. Where such an approach does not result in feasible test requirements, the diagram assumes that the infeasible test requirements are ignored.

3.2.4 Making a Clause Determine a Predicate

So, how does one go about finding values for the minor clauses c_j so that the major clause c_i determines the value of p? The authors are aware of three different methods presented in the literature; we give a direct definitional approach here. Pointers

to the other two, one of which is an algorithmic version of the definitional approach, are given in the bibliographic notes.

For a predicate p with clause (or boolean variable) c, let $p_{c=true}$ represent the predicate p with every occurrence of c replaced by *true* and $p_{c=false}$ be the predicate p with every occurrence of c replaced by *false*. For the rest of this development, we assume no duplicates (that is, p contains only one occurrence of c). Note that neither $p_{c=true}$ nor $p_{c=false}$ contains any occurrences of the clause c. Now we connect the two expressions with an exclusive or:

$$p_c = p_{c=true} \oplus p_{c=false}$$

It turns out that p_c describes the exact conditions under which the value of c determines that of p. That is, if values for the clauses in p_c are chosen so that p_c is true, then the truth value of c determines the truth value of p. If the clauses in p_c are chosen so that p_c evaluates to false, then the truth value of p is independent of the truth value of c. This is exactly what we need to implement the various flavors of active and inactive clause coverage.

As a first example, we try $p = a \vee b$. p_a is, by definition,

$$\begin{aligned} p_a &= p_{a=true} \oplus p_{a=false} \\ &= (true \vee b) \oplus (false \vee b) \\ &= true \oplus b \\ &= \neg b \end{aligned}$$

That is, for the major clause a to determine the predicate p, the only minor clause b must be false. This should make sense intuitively, since the value of a will have an effect on the value of p only if b is false. By symmetry, it is clear that p_b is $\neg a$.

If we change the predicate to $p = a \wedge b$, we get

$$\begin{aligned} p_a &= p_{a=true} \oplus p_{a=false} \\ &= (true \wedge b) \oplus (false \wedge b) \\ &= b \oplus false \\ &= b \end{aligned}$$

That is, we need $b = true$ to make a determine p. By a similar analysis, $p_b = a$.

The equivalence operator is a little less obvious and brings up an interesting point. Consider $p = a \leftrightarrow b$.

$$\begin{aligned} p_a &= p_{a=true} \oplus p_{a=false} \\ &= (true \leftrightarrow b) \oplus (false \leftrightarrow b) \\ &= b \oplus \neg b \\ &= true \end{aligned}$$

That is, for any value of b, a determines the value of p **without regard to the value for** b! This means that for a predicate p, such as this one, where the value of p_c is the constant true, the ICC criteria are infeasible with respect to c. Inactive clause coverage is likely to result in infeasible test requirements when applied to expressions that use the equivalence or exclusive-or operators.

A more general version of this conclusion can be drawn that applies to the ACC criteria as well. If a predicate p contains a clause c such that p_c evaluates to the constant false, the ACC criteria are infeasible with respect to c. The ultimate reason

is that the clause in question is redundant; the predicate can be rewritten without it. While this may sound like a theoretical curiosity, it is actually a very useful result for testers. If a predicate contains a redundant clause, this is a very strong signal that something is wrong with the predicate!

Consider $p = a \wedge b \vee a \wedge \neg b$. This is really just the predicate $p = a$; b is irrelevant. Computing p_b, we get

$$\begin{aligned} p_b &= p_{b=true} \oplus p_{b=false} \\ &= (a \wedge true \vee a \wedge \neg true) \oplus (a \wedge false \vee a \wedge \neg false) \\ &= (a \vee false) \oplus (false \vee a) \\ &= a \oplus a \\ &= false \end{aligned}$$

so it is impossible for b to determine p.

We need to consider how to make clauses determine predicates for a couple of more complicated expressions. For the expression $p = a \wedge (b \vee c)$, we get

$$\begin{aligned} p_a &= p_{a=true} \oplus p_{a=false} \\ &= (true \wedge (b \vee c)) \oplus (false \wedge (b \vee c)) \\ &= (b \vee c) \oplus false \\ &= b \vee c. \end{aligned}$$

This example ends with an undetermined answer, which points out the key difference between CACC and RACC. Three choices of values make $b \vee c$ true, ($b = c =$ true), ($b =$ true, $c =$ false), and ($b =$ false, $c =$ true). For CACC, we could pick one pair of values when a is true and another when a is false. For RACC, we must choose the same pair for both values of a.

The derivation for b and equivalently for c is slightly more complicated:

$$\begin{aligned} p_b &= p_{b=true} \oplus p_{b=false} \\ &= (a \wedge (true \vee c)) \oplus (a \wedge (false \vee c)) \\ &= (a \wedge true) \oplus (a \wedge c) \\ &= a \oplus (a \wedge c) \\ &= a \wedge \neg c \end{aligned}$$

The last step in the simplification shown above may not be immediately obvious. If it is not, try constructing the truth table for $a \oplus (a \wedge c)$. The computation for p_c is equivalent and yields the solution $a \wedge \neg b$.

3.2.5 Finding Satisfying Values

The final step in applying the logic coverage criteria is to choose values that satisfy the criteria. This section shows how to generate values for one example; more cases are explored in the exercises and the application sections later in the chapter. The example is from the first section of the chapter:

$$p = (a \vee b) \wedge c$$

Finding values for predicate coverage is easy and was already shown in Section 3.2. Two test requirements are

$$TR_{PC} = \{p = true, p = false\}$$

and they can be satisfied with the following values for the clauses:

	a	b	c
$p = true$	t	t	t
$p = false$	t	t	f

To run the test cases, we need to refine these truth assignments to create values for clauses a, b, and c. Suppose that clauses a, b, and c were defined in terms of Java program variables as follows:

a	`x < y`, a relational expression for program variables `x` and `y`
b	`done`, a primitive boolean value
c	`list.contains(str)`, for `List` and `String` objects

Thus, the complete expanded predicate is actually

$$p = (x < y \vee done) \wedge list.contains(str)$$

Then the following values for the program variables satisfy the test requirements for predicate coverage.

	a	b	c
$p = true$	x=3 y=5	done = true	list=["Rat," "Cat," "Dog"] str = "Cat"
$p = false$	x=0 y=7	done = true	list=["Red," "White"] str = "Blue"

Note that the values for the program variables need not be the same in a particular test case if the goal is to set a clause to a particular value. For example, clause a is true in both tests, even though program variables x and y have different values.

Values to satisfy clause coverage were also shown in Section 3.2. Six test requirements are

$$TR_{CC} = \{a = true, a = false, b = true, b = false, c = true, c = false\}$$

and they can be satisfied with the following values for the clauses (blank cells represent "don't-care" values):

	a	b	c
$a = true$	t		
$a = false$	f		
$b = true$		t	
$b = false$		f	
$c = true$			t
$c = false$			f

Refining the truth assignments to create values for program variables x, y, *done*, *list*, and *str* is left as an exercise for the reader.

Before proceeding with the other criteria, we first choose values for minor clauses to ensure that the major clauses will determine the value of p. We gave a

method of calculating p_a, p_b, and p_c earlier. The computations for this particular predicate p are left as an exercise. However, the results are

p_a	$\neg b \wedge c$
p_b	$\neg a \wedge c$
p_c	$a \vee b$

Now we can turn to the other clause coverage criteria. The first is combinatorial coverage, requiring all combinations of values for the clauses. In this case, we have eight test requirements, which can be satisfied with the following values:

	a	b	c	$(a \vee b) \wedge c$
1	t	t	t	t
2	t	t	f	f
3	t	f	t	t
4	t	f	f	f
5	f	t	t	t
6	f	t	f	f
7	f	f	t	f
8	f	f	f	f

Recall that general active clause coverage requires that each major clause be true and false and the minor clauses be such that the major clause determines the value of the predicate. Similarly to clause coverage, three pairs of test requirements can be defined:

$$TR_{GACC} = \{(a = true \wedge p_a, a = false \wedge p_a), (b = true \wedge p_b, b = false \wedge p_b), (c = true \wedge p_c, c = false \wedge p_c)\}$$

The test requirements can be satisfied with the following values for the clauses. Note that these can be the same as with clause coverage with the exception that the blank cells from clause coverage are replaced with the values from the determination analysis. In the following (**partial** truth) table, values for major clauses are indicated with upper case letters in boldface.

	a	b	c	p
$a = true \wedge p_a$	**T**	f	t	t
$a = false \wedge p_a$	**F**	f	t	f
$b = true \wedge p_b$	f	**T**	t	t
$b = false \wedge p_b$	f	**F**	t	f
$c = true \wedge p_c$	t	f	**T**	t
$c = false \wedge p_c$	f	t	**F**	f

Note the duplication; the first and fifth rows are identical, and the second and fourth are identical. Thus, only four tests are needed to satisfy GACC.

A different way of looking at GACC considers all of the possible pairs of test inputs for each pair of test requirements. Recall that the active clause coverage criteria always generate test requirements in pairs, with one pair generated for each clause in the predicate under test. To identify these test inputs, we will use the row

numbers from the truth table. Hence, the pair (3, 7) represents the first two tests listed in the table above.

It turns out that (3, 7) is the only pair that satisfies the GACC test requirements with respect to clause a (when a is major), and (5, 7) is the only pair that satisfies the GACC test requirements with respect to clause b. For clause c, the situation is more interesting. Nine pairs satisfy the GACC test requirements for clause c, namely

$$\{(1, 2), (1, 4), (1, 6), (3, 2), (3, 4), (3, 6), (5, 2), (5, 4), (5, 6)\}$$

Recall that correlated active clause coverage requires that each major clause be true and false, the minor clauses be such that the major clause determines the value of the predicate, and the predicate must have both the value true and false. As with GACC, three pairs of test requirements can be defined: For clause a, the pair of test requirements is

$$a = true \wedge p_a \wedge p = x$$
$$a = false \wedge p_a \wedge p = \neg x$$

where x may be either true or false. The point is that p must have a different truth value in the two test cases. We leave the reader to write out the corresponding CACC test requirements with respect to b and c.

For our example predicate p, a careful examination of the pairs of test cases for GACC reveals that p takes on both truth values in each pair. Hence, GACC and CACC are the same for predicate p, and the same pairs of test inputs apply. In the exercises the reader will find predicates where a test pair that satisfies GACC with respect to some clause c turns out not to satisfy CACC with respect to c.

The situation for RACC is quite different, however, in the example p. Recall that restricted active clause coverage is the same as CACC except that it requires the values for the minor clauses c_j to be identical for both assignments of truth values to the major clause, c_i. For clause a, the pair of test requirements that RACC generates is

$$a = true \wedge p_a \wedge b = B \wedge c = C$$
$$a = false \wedge p_a \wedge b = B \wedge c = C$$

for some boolean constants B and C. An examination of the pairs given above for $GACC$ reveals that with respect to clauses a and b, the pairs are the same. So pair (3, 7) satisfies RACC with respect to clause a and pair (5, 7) satisfies RACC with respect to b. However, with respect to c, only three of the pairs satisfy RACC, namely,

$$\{(1, 2), (3, 4), (5, 6)\}$$

This example does leave one question about the different flavors of the ACC criteria, namely, what is the practical difference among them? That is, beyond the subtle difference in the arithmetic, how do they affect practical testers? The real differences do not show up very often, but when they do they can be dramatic and quite annoying.

GACC does not require that predicate coverage be satisfied on the pair of tests for each clause, so use of that flavor may mean we do not test our program as thoroughly as we might like. In practical use, it is easy to construct examples where

GACC is satisfied but predicate coverage is not, when the predicates are very small (one or two terms), but difficult with three or more terms, since for one of the clauses, it is likely that the chosen GACC tests will also be CACC tests.

The restrictive nature of RACC, on the other hand, can sometimes make it hard to satisfy the criterion. This is particularly true when some combinations of clause values are infeasible. Assume that in the predicate used above, the semantics of the program effectively eliminate rows 2, 3, and 6 from the truth table. Then RACC cannot be satisfied with respect to clause *list.contains(str)* (that is, we have infeasible test requirements), but CACC can. The wise reader, (that is, if still awake) will by now realize that Correlated Active Clause Coverage is often the most practical flavor of ACC.

EXERCISES

Section 3.2.

Use predicates (1) through (10) to answer the following questions.

1. $p = a \wedge (\neg b \vee c)$
2. $p = a \vee (b \wedge c)$
3. $p = a \wedge b$
4. $p = a \rightarrow (b \rightarrow c)$
5. $p = a \oplus b$
6. $p = a \leftrightarrow (b \wedge c)$
7. $p = (a \vee b) \wedge (c \vee d)$
8. $p = (\neg a \wedge \neg b) \vee (a \wedge \neg c) \vee (\neg a \wedge c)$
9. $p = a \vee b \vee (c \wedge d)$
10. $p = (a \wedge b) \vee (b \wedge c) \vee (a \wedge c)$
    (a) Identify the clauses that go with predicate p.
    (b) Compute (and simplify) the conditions under which each of the clauses determines predicate p.
    (c) Write the complete truth table for all clauses. Label your rows starting from 1. Use the format in the example underneath the definition of combinatorial coverage in Section 3.2. That is, row 1 should be all clauses true. You should include columns for the conditions under which each clause determines the predicate, and also a column for the predicate itself.
    (d) Identify all pairs of rows from your table that satisfy general active clause coverage (GACC) with respect to each clause.
    (e) Identify all pairs of rows from your table that satisfy correlated active clause coverage (CACC) with respect to each clause.
    (f) Identify all pairs of rows from your table that satisfy restricted active clause coverage (RACC) with respect to each clause.
    (g) Identify all 4-tuples of rows from your table that satisfy general inactive clause coverage (GICC) with respect to each clause. Identify any infeasible GICC test requirements.
    (h) Identify all 4-tuples of rows from your table that satisfy restricted inactive clause coverage (RICC) with respect to each clause. Identify any infeasible RICC test requirements.

11. Refine the GACC, CACC, RACC, GICC, and RICC coverage criteria so that the constraints on the minor clauses are made more formal.
12. (**Challenging!**) Find a predicate and a set of additional constraints so that CACC is infeasible with respect to some clause, but GACC is feasible.

3.3 STRUCTURAL LOGIC COVERAGE OF PROGRAMS

As with graph coverage criteria, the logic coverage criteria apply to programs in a straightforward way. Predicates are derived directly from decision points in the programs (`if`, `case`, and `loop` statements). Although these criteria are difficult to apply when predicates have a large number of clauses, this is often not a problem with programs. The vast majority of predicates in programs have only one clause, and programmers tend to write predicates with a maximum of two or three clauses. It should be clear that when a predicate only has one clause, all of the logic coverage criteria collapse into the same criterion – predicate coverage.

The primary complexity of applying logic coverage to programs has more to do with reachability than with the criteria. That is, a logic coverage criterion imposes test requirements that are related to specific decision points (statements) in the program. Getting values that satisfy those requirements is only part of the problem; getting to the statement is sometimes more difficult. Two issues are associated with getting there. The first is simply that of reachability from Chapter 1; the test case must include values to reach the statement. In small programs (that is, most methods) this problem is not hard, but when applied within the context of an entire arbitrarily large program, satisfying reachability can be enormously complex. The values that satisfy reachability are prefix values in the test case.

The other part of "getting there" can be even harder. The test requirements are expressed in terms of program variables that may be defined locally to the unit or even the statement block being tested. Our test cases, on the other hand, can include values only for inputs to the program that we are testing. Therefore these *internal variables* have to be resolved to be in terms of the input variables. Although the values for the variables in the test requirements should ultimately be a function of the values of the input variables, this relationship may be arbitrarily complex. In fact, this *internal variable* problem is formally undecidable.

Consider an internal variable X that is derived from a table lookup, where the index to the table is determined by a complex function whose inputs are program inputs. To choose a particular value for X, the tester has to work backward from the statement where the decision appears, to the table where X was chosen, to the function, and finally to an input that would cause the function to compute the desired value. If the function includes randomness or is time sensitive, or if the input cannot be controlled by the tester, it may be impossible to satisfy the test requirement with certainty. This controllability problem has been explored in depth in the automatic test data generation literature and will not be discussed in detail here, except to note that this problem is a major reason why the use of program-level logic coverage criteria is usually limited to unit and module testing activities.

The example program in Figures 3.2 and 3.3 is used to illustrate logic coverage on programs.[2] The program is a simple triangle classification program called `TriTyp`. This program (or more accurately, the algorithm) has been used as an example in

```
 1 // Jeff Offutt--Java version Feb 2003
 2 // Classify triangles
 3 import java.io.*;
 4
 5 class trityp
 6 {
 7   private static String[] triTypes = { "", // Ignore 0.
 8          "scalene", "isosceles", "equilateral",
          "not a valid triangle"};
 9   private static String instructions = "This is the ancient
     TriTyp program.\nEnter three integers that represent the
     lengths of the sides of a triangle.\nThe triangle will be
     categorized as either scalene, isosceles, equilateral\n
     or invalid.\n";
10
11 public static void main (String[] argv)
12 {  // Driver program for trityp
13   int A, B, C;
14   int T;
15
16   System.out.println (instructions);
17   System.out.println ("Enter side 1: ");
18   A = getN();
19   System.out.println ("Enter side 2: ");
20   B = getN();
21   System.out.println ("Enter side 3: ");
22   C = getN();
23   T = Triang (A, B, C);
24
25   System.out.println ("Result is: " + triTypes[T]);
26 }
27
28 // ==================================
29 // The main triangle classification method
30 private static int Triang (int Side1, int Side2, int Side3)
31 {
32   int triOut;
33
34   // triOut is output from the routine:
35   //    Triang = 1 if triangle is scalene
36   //    Triang = 2 if triangle is isosceles
37   //    Triang = 3 if triangle is equilateral
38   //    Triang = 4 if not a triangle
39
40   // After a quick confirmation that it's a valid
41   // triangle, detect any sides of equal length
42   if (Side1 <= 0 || Side2 <= 0 || Side3 <= 0)
43   {
44     triOut = 4;
45     return (triOut);
46   }
47
48   triOut = 0;
49   if (Side1 == Side2)
50     triOut = triOut + 1;
51   if (Side1 == Side3)
52     triOut = triOut + 2;
53   if (Side2 == Side3)
54     triOut = triOut + 3;
55   if (triOut == 0)
56   {  // Confirm it's a valid triangle before declaring
57     // it to be scalene
58
```

Figure 3.2. TriTyp – Part A.

```
59        if (Side1+Side2 <= Side3 || Side2+Side3 <= Side1 ||
60            Side1+Side3 <= Side2)
61          triOut = 4;
62        else
63          triOut = 1;
64        return (triOut);
65      }
66
67      // Confirm it's a valid triangle before declaring
68      // it to be isosceles or equilateral
69
70      if (triOut > 3)
71        triOut = 3;
72      else if (triOut == 1 && Side1+Side2 > Side3)
73        triOut = 2;
74      else if (triOut == 2 && Side1+Side3 > Side2)
75        triOut = 2;
76      else if (triOut == 3 && Side2+Side3 > Side1)
77        triOut = 2;
78      else
79        triOut = 4;
80      return (triOut);
81    } // end Triang
82
83    // ==================================
84    // Read (or choose) an integer
85    private static int getN ()
86    {
87      int inputInt = 1;
88      BufferedReader in = new BufferedReader (new InputStreamReader (System.in));
89      String inStr;
90
91      try
92      {
93        inStr    = in.readLine ();
94        inputInt = Integer.parseInt(inStr);
95      }
96      catch (IOException e)
97      {
98        System.out.println ("Could not read input, choosing 1.");
99      }
100     catch (NumberFormatException e)
101     {
102       System.out.println ("Entry must be a number, choosing 1.");
103     }
104
105     return (inputInt);
106   }  // end getN
107
108 }  // end trityp class
```

Figure 3.3. TriTyp – Part B.

the testing literature for many years. As an example, it has several advantages: its purpose is relatively easy to understand, it is small enough to fit in a classroom exercise, and it has a very complicated logic structure that can illustrate most of the concepts. This version of TriTyp is written in Java and was compiled and tested with Sun's JDK Java 1.4.1. Line numbers have been added to allow us to refer to specific decision statements in the text.

Predicates are taken from decision points in the program, including if statements, case/switch statements, for loops, while loops, and do-until loops. This is illustrated with the Triang() method in the TriTyp program. Triang() has the following predicates (line numbers are shown on the left, and the else statements at lines 62 and 78 do not have their own predicates):

```
42: (Side1 <= 0 || Side2 <= 0 || Side3 <= 0)
49: (Side1 == Side2)
51: (Side1 == Side3)
53: (Side2 == Side3)
55: (triOut == 0)
59: (Side1+Side2 <= Side3 || Side2+Side3 <= Side1 ||
     Side1+Side3 <= Side2)
70: (triOut > 3)
72: (triOut == 1 && Side1+Side2 > Side3)
74: (triOut == 2 && Side1+Side3 > Side2)
76: (triOut == 3 && Side2+Side3 > Side1)
```

The TriTyp program has three inputs, which are read into variables A, B, and C in the main program method and then passed to the formal parameters Side1, Side2, and Side3 in Triang(). The rest of this section illustrates how to satisfy the logic coverage criteria on TriTyp. Before addressing the actual criteria, it is first necessary to analyze the predicates to find values that will reach the predicates (the reachability problem) and to understand how to assign particular values to the variable triOut (the internal variable problem).

In the Tables 3.1 through 3.3, Side1, Side2, and Side3 are abbreviated as S1, S2, and S3 to save space. First we consider reachability. The predicate on line 42 is

Table 3.1. Reachability for Triang predicates

```
42: True
49: P1 = (S1 > 0 && S2 > 0 && S3 > 0)
51: P1
53: P1
55: P1
59: P1 && (triOut == 0)
62: P1 && (triOut == 0)
        && (S1+S2 > S3) && (S2+S3 > S1) && (S1+S3 > S2)
70: P1 && (triOut != 0)
72: P1 && (triOut != 0) && (triOut <= 3)
74: P1 && (triOut != 0) && (triOut <= 3)
        && ((triOut != 1) || (S1+S2 <= S3))
76: P1 && (triOut != 0) && (triOut <= 3)
        && ((triOut != 1) || (S1+S2 <= S3))
        && ((triOut != 2) || (S1+S3 <= S2))
78: P1 && (triOut != 0) && (triOut <= 3)
        && ((triOut != 1) || (S1+S2 <= S3))
        && ((triOut != 2) || (S1+S3 <= S2))
        && ((triOut != 3) || (S2+S3 <= S1))
```

always reached whenever Triang() is called, so its predicate is true, as shown in the first line of Table 3.1. Line 49 is reached only if the predicate on line 42 is false, so its reachability predicate is the inverse, S1 > 0 && S2 > 0 && S3 > 0. This predicate is given the label P1 and referred to in subsequent reachability predicates. The rest of the predicates are found in the similar manner, by negating the predicate on the previous edge.

Note that several predicates in Table 3.1 reference the variable tri0ut, which is a local (internal) variable assigned a value at lines 44, 48, 50, 52, 54, 61, 63, 71, 73, 75, 77, and 79. So the next step in generating tests is to discover how to assign specific values to tri0ut. At line 55, tri0ut has a value in the range (0..6) inclusive, as assigned in the previous statements. By applying the predicates in Table 3.1 and tracing the assignments, we can determine the following rules for tri0ut:

tri0ut	Rules for determining tri0ut
0	S1 != S2 && S1 != S3 && S2 != S3
1	S1 == S2 && S1 != S3 && S2 != S3
2	S1 != S2 && S1 == S3 && S2 != S3
3	S1 != S2 && S1 != S3 && S2 == S3
4	S1 == S2 && S1 != S3 && S2 == S3
5	S1 != S2 && S1 == S3 && S2 == S3
6	S1 == S2 && S1 == S3 && S2 == S3

The predicates for tri0ut equal to 4 and 5 are contradictions, so it cannot have those values after line 55. These values can be used to reduce and simplify the predicates in Table 3.1, resulting in the predicates shown in Table 3.2. These predicates

Table 3.2. **Reachability for Triang predicates – reduced by solving for tri0ut**

```
42: True
49: P1 = (S1 > 0 && S2 > 0 && S3 > 0)
51: P1
53: P1
55: P1
59: P1 && (S1 != S2 && S1 != S3 && S2 != S3)          (tri0ut == 0)
62: P1 && (S1 != S2 && S1 != S3 && S2 != S3)          (tri0ut == 0)
      && (S1+S2 > S3) && (S2+S3 > S1) && (S1+S3 > S2)
70: P1 && P2 = (S1 == S2 || S1 == S3 || S2 == S3)     (tri0ut != 0)
72: P1 && P2 && P3 = (S1!=S2 || S1!=S3 || S2!=S3)     (tri0ut <= 3)
74: P1 && P2 && P3 && (S1 != S2 || S1 + S2 <= S3)
76: P1 && P2 && P3 && (S1 != S2 || S1 + S2 <= S3)
      && ((S1 != S3) || (S1+S3 <= S2))
78: P1 && P2 && P3 && (S1 != S2 || S1 + S2 <= S3)
      && ((S1 != S3) || (S1+S3 <= S2))
      && ((S2 != S3) || (S2+S3 <= S1))
```

Table 3.3. Predicate coverage for Triang

	Predicate	True				False			
		A	B	C	EO	A	B	C	EO
p42:	$(S1 \leq 0 \vee S2 \leq 0 \vee S3 \leq 0)$	0	0	0	4	1	1	1	3
p49:	$(S1 == S2)$	1	1	1	3	1	2	2	3
p51:	$(S1 == S3)$	1	1	1	3	1	2	2	2
p53:	$(S2 == S3)$	1	1	1	3	2	1	2	2
p55:	$(triOut == 0)$	1	2	3	4	1	1	1	3
p59:	$(S1 + S2 \leq S3 \ \vee$ $S2 + S3 \leq S1 \ \vee$ $S1 + S3 \leq S2)$	1	2	3	4	2	3	4	1
p70:	$(triOut > 3)$	1	1	1	3	2	2	3	2
p72:	$(triOut == 1 \wedge S1 + S2 > S3)$	2	2	3	2	2	2	4	4
p74:	$(triOut == 2 \wedge S1 + S3 > S2)$	2	3	2	2	2	4	2	4
p76:	$(triOut == 3 \wedge S2 + S3 > S1)$	3	2	2	2	4	2	2	4

are beginning to look complicated, but they contain a lot of redundancy. Two more named formulas have been introduced, `P2 = (S1 == S2 || S1 == S3 || S2 == S3)` and `P3 = (S1 != S2 || S1 != S3 || S2 != S3)`. Table 3.2 shows the predicates that will guarantee reachability for the given statement numbers. These are then used to find satisfying values for the logic criteria.

Finding values to satisfy **predicate coverage** for the predicate on line 42 in `TriTyp` is straightforward. Any one of the three variables can be given a value of 0 or less for the true case, and they all three have to be 1 or greater for the false case.

Predicates 49, 51, and 53 are similar. They are true if two of the sides have the same length and false if the values are of different lengths. Predicate 55 demonstrates the internal variable problem. *triOut* is an internal variable that has definitions on statements 48, 50, 52, and 54. Algebraic analysis can show that *triOut* is 0 only if neither of the predicates at lines 49, 51 or 53 is true. A semantic generalization of this analysis is that for *triOut* to be 0 (predicate 55 is true), all three sides have to have different lengths.

The other predicates are resolved in similar ways, with the internal variable *triOut* being resolved for values using the previous analysis. Values to satisfy predicate coverage are shown in Table 3.3. For each predicate, the inputs to guarantee the predicate be true and then false are given, along with the expected output (EO). It is generally safer to postpone choosing those values until reachability is determined.

It should be obvious from this example that predicate coverage on programs is simply another way to formulate the edge coverage criterion. It is not necessary to draw a graph for the logic criteria, but the control flow graph can help find values for reachability.

Previously we said that selection of values for "don't care" inputs should be postponed until reachability is determined. This is because of potential interactions with the requirements for reachability and the selection of values. That is, some inputs may be "don't care" for the test requirements, but may need specific values to reach

Table 3.4. Clause coverage for Triang

	Clause	True A	True B	True C	True EO	False A	False B	False C	False EO
p42:	$(S1 \leq 0)$	0	1	1	4	1	1	1	3
	$(S2 \leq 0)$	1	0	1	4	1	1	1	3
	$(S3 \leq 0)$	1	1	0	4	1	1	1	3
p59:	$(S1 + S2 \leq S3)$	2	3	6	4	2	3	4	1
	$(S2 + S3 \leq S1)$	6	2	3	4	2	3	4	1
	$(S1 + S3 \leq S2)$	2	6	3	4	2	3	4	1
p72:	$(triOut = 1)$	2	2	3	2	2	3	2	2
	$(S1 + S2 > S3)$	2	2	3	2	2	2	5	4
p74:	$(triOut = 2)$	2	3	2	2	3	2	2	2
	$(S1 + S3 > S2)$	2	3	2	2	2	5	2	4
p76:	$(triOut = 3)$	3	2	2	2	1	2	1	4
	$(S2 + S3 > S1)$	3	2	2	2	5	2	2	4

the decision. Thus, if we select values too early, it may become impossible to satisfy reachability.

The values needed to satisfy the other criteria are the same for the predicates that have only one clause (p49, p51, p53, p55, and p70). Thus, we consider **clause coverage** only for the predicates that have more than one clause.

The clauses for predicate 42 are simple, and simply require a value for one of the variables. The other values can be chosen arbitrarily. The other predicates are a bit more complicated, but again can be solved with simple algebra. Consider the first clause in predicate 59, $S1 + S2 \leq S3$. If we choose values for $S1$ and $S2$ as $(S1 = 2, S2 = 3)$, then $S1 + S2 = 5$, so $S3$ should be at least 5 for the true case. The same values can be used for $S1$ and $S2$ for the false case, and $S3$ can be 4. The first clause in predicate 72 has an additional complexity because it involves the internal variable *triOut*. The logic of *Triang*() reveals that $triOut = 1$ if and only if $S1 = S2$, and neither $S1$ nor $S2$ equals $S3$. So the true case can be satisfied with $(2, 2, 3)$ and the false case can be satisfied by making the values of the variables different $(2, 3, 4)$.

Values to satisfy Clause Coverage are shown in Table 3.4.

Rather than going through all of the other criteria, we just focus on **correlated active clause coverage**. Because each predicate involves only one operator (|| or &&) the determination analysis is straightforward. For ||, the minor clauses must be false, and for &&, the minor clauses must be true. This example does bring up one additional complication. In p59, each clause is distinct, yet the individual clauses contain the same variables. This can sometimes make it difficult (and even impossible) to find values that satisfy CACC.

For example, consider the major clause $(S1 + S2 \leq S3)$. For it to determine the predicate, the two minor clauses $(S2 + S3 \leq S1)$ and $(S1 + S3 \leq S2)$ must both be false. If the test case $(0, 0, 0)$ is chosen to make $(S1 + S2 \leq S3)$ true, then all three clauses are true and $(S1 + S2 \leq S3)$ does not determine the predicate. (In the particular case of p59, however, because of the algebra of the requirements,

Table 3.5. Correlated active clause coverage for Triang

Predicate		Clauses			A	B	C	EO
p42:	$(S1 \leq 0 \vee S2 \leq 0 \vee S3 \leq 0)$	T	f	f	0	1	1	4
		F	f	f	1	1	1	3
		f	T	f	1	0	1	4
		f	f	T	1	1	0	4
p59:	$(S1 + S2 \leq S3 \vee$	T	f	f	2	3	6	4
	$S2 + S3 \leq S1 \vee$	F	f	f	2	3	4	1
	$S1 + S3 \leq S2)$	f	T	f	6	2	3	4
		f	f	T	2	6	3	4
p72:	$(triOut == 1 \wedge S1 + S2 > S3)$	T	t	–	2	2	3	2
		F	t	–	2	3	3	2
		t	F	–	2	2	5	4
p74:	$(triOut == 2 \wedge S1 + S3 > S2)$	T	t	–	2	3	2	2
		F	t	–	2	3	3	2
		t	F	–	2	5	2	4
p76:	$(triOut == 3 \wedge S2 + S3 > S1)$	T	t	–	3	2	2	2
		F	t	–	3	6	3	4
		t	F	–	5	2	2	4

(0, 0, 0) is the only test that has this problem.) Values to satisfy CACC are shown in Table 3.3.

3.3.1 Predicate Transformation Issues

ACC criteria are considered to be expensive for testers, and attempts have been made to reduce the cost. One approach is to rewrite the program to eliminate multiclause predicates, thus reducing the problem to branch testing. A conjecture is that the resulting tests will be equivalent to ACC. However, we explicitly advise against this approach for two reasons. One, the resulting rewritten program may have substantially more complicated control structure than the original (including repeated statements), thus endangering both reliability and maintainability. Second, as the following examples demonstrate, the transformed program may not require tests that are equivalent to the tests for ACC on the original program.

Consider the following program segment, where a and b are arbitrary boolean clauses and S1 and S2 are arbitrary statements. S1 and S2 could be single statements, block statements, or function calls.

```
if (a && b)
   S1;
else
   S2;
```

The CACC criterion requires the test specifications (t, t), (t, f), and (f, t) for the predicate $a \wedge b$. However, if the program segment is transformed into the following functionally equivalent structure:

```
if (a)
{
  if (b)
    S1;
  else
    S2;
}
else
  S2;
```

the predicate coverage criterion requires three tests: (t, t) to reach statement S1, (t, f) to reach the first occurrence of statement S2, and either (f, f) or (f, t) to reach the second occurrence of statement S2. Choosing (t, t), (t, f), and (f, f) means that our tests do **not** satisfy CACC in that they do not allow a to determine fully the predicate's value. Moreover, the duplication of S2 in the above example has been taught to be poor programming for years, because of the potential for mistakes when duplicating code.

A larger example reveals the flaw even more clearly. Consider the simple program segment

```
if ((a && b) || c)
  S1;
else
  S2;
```

A straightforward rewrite of this program fragment to remove the multiclause predicate results in this complicated ugliness:

```
if (a)
  if (b)
    if (c)
      S1;
    else
      S1;
  else
    if (c)
      S1;
    else
      S2;
else
 if (b)
   if (c)
     S1;
```

```
    else
      S2;
  else
    if (c)
      S1;
    else
      S2;
```

This fragment is cumbersome in the extreme, and likely to be error-prone. Applying the predicate coverage criterion to this would be equivalent to applying combinatorial coverage to the original predicate. A reasonably clever programmer (or good optimizing compiler) would simplify it as follows:

```
if (a)
  if (b)
    S1;
  else
    if (c)
      S1;
    else
      S2;
else
  if (c)
    S1;
  else
    S2;
```

This fragment is still much harder to understand than the original. Try to imagine a maintenance programmer trying to change this thing!

The following table illustrates truth assignments that can be used to satisfy CACC for the original program segment and predicate testing for the modified version. An 'X' under CACC or predicate indicates that truth assignment is used to satisfy the criterion for the appropriate program fragment. Clearly, predicate coverage on an equivalent program is not the same as CACC testing on the original. Predicate coverage on this modified program does not subsume CACC, and CACC does not subsume predicate coverage.

	a	b	c	$((a \wedge b) \vee c)$	CACC	Predicate
1	t	t	t	T		X
2	t	t	f	T	X	
3	t	f	t	T	X	X
4	t	f	f	F	X	X
5	f	t	t	T		X
6	f	t	f	F	X	
7	f	f	t	T		
8	f	f	f	F		X

EXERCISES

Section 3.3.

1. Answer the following questions for the method checkIt() below:

```
public static void checkIt (boolean a, boolean b, boolean c)
{
  if (a && (b || c))
  {
     System.out.println ("P is true");
  }
  else
  {
     System.out.println ("P isn't true");
  }
}
```

   - Transform *checkIt()* to *checkItExpand()*, a method where each *if* statement tests exactly one boolean variable. Instrument *checkItExpand()* to record which edges are traversed. ("print" statements are fine for this.)
   - Derive a GACC test set *T1* for *checkIt()*. Derive an edge coverage test set *T2* for *checkItExpand()*. Build *T2* so that it does **not** satisfy GACC on the predicate in *checkIt()*.
   - Run both *T1* and *T2* on both *checkIt()* and *checkItExpand()*.
2. Answer the following questions for the method twoPred() below:

```
public String twoPred (int x, int y)
{
   boolean z;

   if (x < y)
      z = true;
   else
      z = false;

   if (z && x+y == 10)
      return "A";
   else
      return "B";
}
```

   - Identify test inputs for *twoPred()* that achieve Restricted Active Clause Coverage (RACC).
   - Identify test inputs for *twoPred()* that achieve Restricted Inactive Clause Coverage (RICC).
3. Answer the following questions for the program fragments below:

```
fragment P:
  if (A || B || C)
  {
    m();
  }
  return;
```

```
fragment Q:
  if (A)
  {
    m();
    return;
  }
  if (B)
  {
    m();
    return;
  }
  if (C)
  {
    m();
  }
```

- Give a GACC test set for fragment P. (Note that GACC, CACC, and RACC yield identical test sets for this example.)
- Does the GACC test set for fragment P satisfy edge coverage on fragment Q?
- Write down an edge coverage test set for fragment Q. Make your test set include as few tests from the GACC test set as possible.

4. (**Challenging!**) For the TriTyp program, complete the test sets for the following coverage criteria by filling in the "don't care" values, ensuring reachability, and deriving the expected output. Download the program, compile it, and run it with your resulting test cases to verify correct outputs.
   - Predicate coverage (PC)
   - Clause coverage (CC)
   - Combinatorial coverage (CoC)
   - Correlated active clause coverage (CACC)
5. Repeat the prior exercise, but for the TestPat program in Chapter 2.
6. Repeat the prior exercise, but for the Quadratic program in Chapter 2.

3.4 SPECIFICATION-BASED LOGIC COVERAGE

Software specifications, both formal and informal, appear in a variety of forms and languages. They almost invariably include logical expressions, allowing the logic coverage criteria to be applied. We start by looking at their application to simple preconditions on methods.

Programmers often include preconditions as part of their methods. The preconditions are sometimes written as part of the design, and sometimes added later as documentation. Specification languages typically make preconditions explicit with the goal of analyzing the preconditions in the context of an invariant. A tester may consider developing the preconditions specifically as part of the testing process if preconditions do not exist. For a variety of reasons, including defensive programming and security, transforming preconditions into exceptions is common practice. In brief, preconditions are common and rich sources of predicates in specifications,

```
public static int cal (int month1, int day1, int month2,
                       int day2, int year)
{
//*************************************************************
// Calculate the number of Days between the two given days in
// the same year.
// preconditions : day1 and day2 must be in same year
//              1 <= month1, month2 <= 12
//              1 <= day1, day2 <= 31
//              month1 <= month2
//              The range for year: 1 ... 10000
//*************************************************************
  int numDays;

  if (month2 == month1) // in the same month
    numDays  = day2 - day1;
  else
  {
    // Skip month 0.
    int daysIn[] = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    // Are we in a leap year?
    int m4 = year % 4;
    int m100 = year % 100;
    int m400 = year % 400;
    if ((m4 != 0) || ((m100 == 0) && (m400 != 0)))
      daysIn[2] = 28;
    else
      daysIn[2] = 29;

    // start with days in the two months
    numDays = day2 + (daysIn[month1] - day1);

    // add the days in the intervening months
    for (int i = month1 + 1; i <= month2-1; i++)
      numDays = daysIn[i] + numDays;
  }
  return (numDays);
}
```

Figure 3.4. Calendar method.

and so we focus on them here. Of course, other specification constructs, such as postconditions and invariants, also are rich sources of complex predicates.

Consider the cal method in Figure 3.4. The method lists explicit preconditions in natural language. These can be translated into predicate form as follows:

$$month1 >= 1 \wedge month1 <= 12 \wedge month2 >= 1 \wedge month2 <= 12 \wedge month1 <= month2$$
$$\wedge\, day1 >= 1 \wedge day1 <= 31 \wedge day2 >= 1 \wedge day2 <= 31 \wedge year >= 1 \wedge year <= 10000$$

The comment about $day1$ and $day2$ being in the same year can be safely ignored, because that prerequisite is enforced syntactically by the fact that only one parameter appears for $year$. It is probably also clear that these preconditions are not complete. Specifically, a day of 31 is valid only for some months. This requirement should be reflected in the specifications or in the program.

This predicate has a very simple structure. It has eleven clauses (which sounds like a lot!) but the only logical operator is "and." Satisfying predicate coverage for cal() is simple – all clauses need to be true for the true case and at least one clause needs to be false for the false case. So ($month1 = 4$, $month2 = 4$, $day1 = 12$, $day2 = 30$, $year = 1961$) satisfies the true case, and the false case is satisfied by violating the clause $month1 <= month2$, with ($month1 = 6$, $month2 = 4$, $day1 = 12$, $day2 = 30$, $year = 1961$). Clause coverage requires all clauses to be true and false.

Table 3.6. Correlated active clause coverage for cal() preconditions

		$m1 \geq 1$	$m1 \leq 12$	$m2 \geq 1$	$m2 \leq 12$	$m1 \leq m2$	$d1 \geq 1$	$d1 \leq 31$	$d2 \geq 1$	$d2 \leq 31$	$y \geq 1$	$y \leq 10000$
1.	$m1 \geq 1 = T$	T	t	t	t	t	t	t	t	t	t	t
2.	$m1 \geq 1 = F$	F	t	t	t	t	t	t	t	t	t	t
3.	$m1 \leq 12 = F$	t	F	t	t	t	t	t	t	t	t	t
4.	$m2 \geq 1 = F$	t	t	F	t	t	t	t	t	t	t	t
5.	$m2 \leq 12 = F$	t	t	t	F	t	t	t	t	t	t	t
6.	$m1 \leq m2 = F$	t	t	t	t	F	t	t	t	t	t	t
7.	$d1 \geq 1 = F$	t	t	t	t	t	F	t	t	t	t	t
8.	$d1 \leq 31 = F$	t	t	t	t	t	t	F	t	t	t	t
9.	$d2 \geq 1 = F$	t	t	t	t	t	t	t	F	t	t	t
10.	$d2 \leq 31 = F$	t	t	t	t	t	t	t	t	F	t	t
11.	$y \geq 1 = F$	t	t	t	t	t	t	t	t	t	F	t
12.	$y \leq 10000 = F$	t	t	t	t	t	t	t	t	t	t	F

We might try to satisfy this requirement with only two tests, but some clauses are related and cannot both be false at the same time. For example, $month1$ cannot be less than 1 and greater than 12 at the same time. The true test for predicate coverage allows all clauses to be true, then we use the following tests to make each clause false: ($month1 = -1$, $month2 = -2$, $day1 = 0$, $day2 = 0$, $year = 0$) and ($month1 = 13$, $month2 = 14$, $day1 = 32$, $day2 = 32$, $year = 10500$).

We must first find how to make each clause determine the predicate to apply the ACC criteria. This turns out to be simple with disjunctive normal form predicates–all we have to do is make each minor clause true. To find the remaining tests, each other clause is made to be false in turn. Therefore, CACC (also RACC and GACC) is satisfied by the tests that are specified in Table 3.6. (To save space, we use abbreviations of the variable names.)

EXERCISES

Section 3.4.

Consider the `remove()` method from the Java `Iterator` interface. The `remove()` method has a complex precondition on the state of the `Iterator`, and the programmer can choose to detect violations of the precondition and report them as `IllegalStateException`.

1. Formalize the precondition.
2. Find (or write) an implementation of an `Iterator`. The Java `Collection` classes are a good place to search.
3. Develop and run CACC tests on the implementation.

3.5 LOGIC COVERAGE OF FINITE STATE MACHINES

Chapter 2 discussed the application of graph coverage criteria to finite state machines (FSMs). Recall that FSMs are graphs with nodes that represent states and edges that represent transitions. Each transition has a pre-state and a post-state. FSMs usually model behavior of the software and can be more or less formal and precise, depending on the needs and inclinations of the developers. This text views FSMs in the most generic way, as graphs. Differences in notations are considered only in terms of the effect they have on applying the criteria.

The most common way to apply logic coverage criteria to FSMs is to use logical expressions from the transitions as predicates. In the Elevator example in Chapter 2, the trigger and thus the predicate is $openButton = pressed$. Tests are created by applying the criteria from Section 3.2 to these predicates.

Consider the example in Figure 3.5. This FSM models the behavior of the memory seat in a car (Lexus 2003 ES300). The memory seat has two configurations for two separate drivers and controls the side mirrors (`sideMirrors`), the vertical height of the seat (`seatBottom`), the horizontal distance of the seat from the steering wheel (`seatBack`), and the lumbar support (`lumbar`). The intent is to remember the configurations so that the drivers can conveniently switch configurations with the press of a button. Each state in the figure has a number for efficient reference.

Figure 3.5. FSM for a memory car seat – Lexus 2003 ES300.

The initial state of the FSM is whichever configuration it was in when the system was last shut down, either Driver 1, Driver 2, or Modified Configuration. The drivers can modify the configuration by changing one of the four controls; changing the side mirrors, moving the seat backwards or forwards, raising or lowering the seat, or modifying the lumbar support (triggering events). These controls work only if the ignition is on (a guard). The driver can also change to the other configuration by pressing either `Button1` or `Button2` when the ignition is on. In these cases, the guards allow the configuration to be changed only if the `Gear` is in `Park` or the ignition is off. These are *safety constraints*, because it would be dangerous to allow the driver's seat to go flying around when the car is moving.

When the driver changes one of the controls, the memory seat is put into the modified configuration state. The new state can be saved by simultaneously pressing the `Reset` button and either `Button1` or `Button2` when the ignition is on. The new configuration is saved permanently when the ignition is turned off.

This type of FSM provides an effective model for testing software, although several issues must be understood and dealt with when creating predicates and then test values. Guards are not always explicitly listed as conjuncts, but they are conjuncts in effect and so should be combined with the triggers using the AND operator. In some specification languages, most notably SCR, the triggers actually imply two values. In SCR, if an event is labeled as triggering, it means that the value of the resulting expression must explicitly change. This implies two values, a before value and an

Table 3.7. Predicates from memory seat example

Pre-state	Post-state	Predicate
1	2	$Button2 \wedge (Gear = Park \vee ignition = off)$
1	3	$sideMirors \wedge ignition = on$
1	3	$seatButton \wedge ignition = on$
1	3	$lumbar \wedge ignition = on$
1	3	$seatBack \wedge ignition = on$
2	1	$Button1 \wedge (Gear = Park \vee ignition = off)$
2	3	$sideMirors \wedge ignition = on$
2	3	$seatButton \wedge ignition = on$
2	3	$lumbar \wedge ignition = on$
2	3	$seatBack \wedge ignition = on$
3	1	$Button1 \wedge (Gear = Park \vee ignition = off)$
3	2	$Button2 \wedge (Gear = Park \vee ignition = off)$
3	4	$Reset \wedge Button1 \wedge ignition = on$
3	5	$Reset \wedge Button2 \wedge ignition = on$
4	1	$ignition = off$
4	3	$sideMirors \wedge ignition = on$
4	3	$seatButton \wedge ignition = on$
4	3	$lumbar \wedge ignition = on$
4	3	$seatBack \wedge ignition = on$
5	2	$ignition = off$
5	3	$sideMirors \wedge ignition = on$
5	3	$seatButton \wedge ignition = on$
5	3	$lumbar \wedge ignition = on$
5	3	$seatBack \wedge ignition = on$

after value, and is modeled by introducing a new variable. For example, in the memory seat example, the transition from `New Configuration Driver 1` to `Driver 1 Configuration` is taken when the ignition is turned off. If that is a triggering transition in the SCR sense, then the predicate needs to have two parts: $ignition = on \wedge ignition' = off$. *ignition'* is the after value.

The transitions from `Modified Configuration` to the two `New Configuration` states demonstrate another issue. The two buttons `Reset` and `Button1` (or `Button2`) must be pressed **simultaneously**. In practical terms for this example, we would like to test for what happens when one button is pressed slightly prior to the other. Unfortunately, the mathematics of logical expressions used in this chapter do not have an explicit way to represent this requirement, thus it is not handled explicitly. The two buttons are connected in the predicate with the AND operator. In fact, this is a simple example of the general problem of timing, and needs to be addressed in the context of real-time software.

The predicates for the memory seat example are in Table 3.7 (using the state numbers from Figure 3.5).

The tests to satisfy the various criteria are fairly straightforward and are left to the exercises. Several issues must be addressed when choosing values for test cases. The first is that of reachability; the test case must include prefix values to reach the pre-state. For most FSMs, this is just a matter of finding a path from an initial state to the pre-state (using a depth first search), and the predicates associated with the

transitions are solved to produce inputs. The memory seat example has three initial states, and the tester cannot control which one is entered because it depends on the state the system was in when it was last shut down. In this case, however, an obvious solution presents itself. We can begin every test by putting the `Gear` in park and pushing `Button 1` (part of the prefix). If the system is in the `Driver 2` or the `Modified Configuration` state, these inputs will cause the system to transition to the `Driver 1` state. If the system is in the `Driver 1` state, these inputs will have no effect. In all three cases, the system will effectively start in the `Driver 1` state.

Some FSMs also have exit states that must be reached with postfix values. Finding these values is essentially the same as finding prefix values; that is, finding a path from the post-state to a final state. The memory seat example does not have an exit state, so this step can be skipped. We also need a way to see the results of the test case (verification values). This might be possible by giving an input to the program to print the current state, or causing some other output that is dependent on the state. The exact form and syntax this takes depends on the implementation, and so it cannot be finalized until the input-output behavior syntax of the software is designed.

One major advantage of this form of testing is determining the expected output. It is simply the post-state of the transition for the test case values that cause the transition to be true, and the pre-state for the test case values that cause the transition to be false (the system should remain in the current state). The only exception to this rule is that occasionally a false predicate might coincidentally be a true predicate for another transition, in which case the expected output should be the post-state of the alternate transition. This situation can be recognized automatically. Also, if a transition is from a state back to itself, then the pre-state and the post-state are the same and the expected output is the same whether the transition is true or false.

The final problem is that of converting a test case (composed of prefix values, test case values, postfix values, and expected output) into an executable test script. The potential problem here is that the variable assignments for the predicates must be converted into inputs to the software. This has been called the *mapping problem* with FSMs and is analogous to the internal variable problem of Section 3.3. Sometimes this step is a simple syntactic rewriting of predicate assignments (`Button1` to program input *button1*). Other times, the input values can be directly encoded as method calls and embedded into a program (for example, `Button1` becomes *pressButton1*()). At other times, however, this problem is much greater and can involve turning seemingly small inputs at the FSM modeling level into long sequences of inputs or method calls. The exact situation depends on the software implementation; thus a general solution to this problem is elusive at best.

EXERCISES

Section 3.5.

1. For the `Memory Seat` finite state machine, complete the test sets for the following coverage criteria by satisfying the predicates, ensuring reachability, and computing the expected output.
   - Predicate coverage
   - Correlated active clause coverage
   - General inactive clause coverage

2. Redraw Figure 3.5 to have fewer transitions, but more clauses. Specifically, nodes 1, 2, 4, and 5 each has four transitions to node 3. Rewrite these transitions to have only one transition from each of nodes 1, 2, 4, and 5 to node 3, and the clauses are connected by ORs. Then derive tests to satisfy CACC for the resulting predicates. How do these tests compare with the tests derived from the original graph?
3. Consider the following deterministic finite state machine:

Current State	Condition	Next State
Idle	$a \vee b$	Active
Active	$a \wedge b$	Idle
Active	$\neg b$	WindDown
WindDown	a	Idle

   (a) Draw the finite state machine.
   (b) This machine does not specify which conditions cause a state to transition back to itself. However, these conditions can be derived from the existing conditions. Derive the conditions under which each state will transition back to itself.
   (c) Find CACC tests for each transition from the Active state.
4. Pick a household appliance such as a watch, calculator, microwave, VCR, clock-radio, or programmable thermostat. Draw the FSM that represents your appliance's behavior. Derive tests to satisfy predicate coverage, correlated active clause coverage, and general inactive clause coverage.
5. Implement the memory seat FSM. Design an appropriate input language to your implementation and turn the tests derived for question 1 into test scripts. Run the tests.

3.6 DISJUNCTIVE NORMAL FORM CRITERIA

In this section, we revisit the testing of boolean expressions. Instead of focusing on each clause by itself, we look at the structure of the predicate as expressed in a disjunctive normal form (DNF) representation.

A *literal* is a clause or the negation of a clause. A *term* is a set of literals connected by logical ANDs. A *Disjunctive Normal Form (DNF)* predicate is a set of terms connected by logical ORs. Terms in DNF predicates are also called *implicants*, because if a term is true, the entire predicate is also true.

For example, this predicate is in disjunctive normal form:

$$(a \wedge \neg c) \vee (b \wedge \neg c)$$

but this (equivalent) one is not:

$$(a \vee b) \wedge \neg c$$

In general, the DNF representation of a predicate is not unique. For example, the above predicate can be rewritten in the following DNF form:

$$(a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge \neg c) \vee (\neg a \wedge b \wedge \neg c)$$

In this section, we follow convention for DNF representations and use adjacency for the $\wedge$ operator and an overstrike for the negation operator. This approach makes the sometimes long expressions easier to read. So, the last DNF predicate above will be written

$$ab\bar{c} \vee a\bar{b}\bar{c} \vee \bar{a}b\bar{c}$$

For our purposes, the interesting thing about the DNF representations is the test criteria that go with them. One way of testing with respect to DNF representations is to assign values to clauses so that each implicant in the DNF representation is satisfied on at least one test. Notice that all of these tests result in the predicate evaluating to true, causing the problem that we never test the false case. We address this problem by formulating a DNF expression for the negation of the predicate in question, and evaluating tests for the negated predicate with the same coverage criteria used for the predicate itself. These ideas are enough to define our first DNF coverage criterion:

> CRITERION **3.20 Implicant Coverage (IC):** *Given DNF representations of a predicate f and its negation $\bar{f}$, for each implicant in f and $\bar{f}$, TR contains the requirement that the implicant evaluate to true.*

As an example of IC, consider the following DNF expression for a predicate f in three clauses.

$$f(a,\ b,\ c)\ =\ ab \vee b\bar{c}$$

Its negation can be computed as follows:

$$\begin{aligned}
\bar{f}(a,\ b,\ c) &= \overline{ab \vee b\bar{c}} \\
&= \overline{ab} \wedge \overline{b\bar{c}} \\
&= (\bar{a} \vee \bar{b}) \wedge (\bar{b} \vee c) \\
&= \bar{a}\bar{b} \vee \bar{a}c \vee \overline{bb} \vee \bar{b}c \\
&= (\bar{a}\bar{b} \vee \overline{bb}) \vee \bar{b}c \vee \bar{a}c \\
&= (\bar{b} \vee \bar{b}c) \vee \bar{a}c \\
&= \bar{b} \vee \bar{a}c
\end{aligned}$$

Collectively, f and $\bar{f}$ have a total of four implicants:

$$\{ab,\ b\bar{c},\ \bar{b},\ \bar{a}c\}$$

An obvious but simple way to generate tests for these four implicants would be to choose one test for each. However, they can be satisfied with fewer tests. Consider the following table, which indicates the truth assignments required for each of the four implicants.

	a	b	c		
1) ab	T	T		a	b
2) $b\bar{c}$		T	F	b	$\bar{c}$
3) $\bar{b}$		F		$\bar{b}$	
4) $\bar{a}c$	F		T	$\bar{a}$	c

The first and second row can be satisfied simultaneously, as can the third and fourth. Thus only two tests are needed to satisfy IC for this example:

$$T_1 = \{TTF,\ FFT\}$$

Note that IC subsumes predicate coverage, but will not necessarily subsume any of the ACC criteria.

A problem with IC is that tests might be chosen so that a single test satisfies multiple implicants. Indeed, this is how the two element test set T_1 above was chosen. Although this lets testers minimize the size of test suites, it is a bad thing from the perspective of testing the unique contributions that each implicant might bring to a predicate. Thus we introduce a method to force a kind of "independence" of the implicant.

The first step is to obtain a DNF form where each implicant can be satisfied without satisfying any other implicant. Fortunately, standard approaches already exist that can be used. A *proper subterm* of an implicant is an implicant with one or more subterms removed. For example, proper subterms of abc are ab and b. A *prime implicant* is an implicant such that no proper subterm of the implicant is also an implicant of the same predicate. That is, in a prime implicant, it is not possible to remove a term without changing the value of the predicate. For example, in the following reformulation of the previous example

$$f(a, b, c) = abc \vee ab\bar{c} \vee b\bar{c}$$

abc is not a prime implicant, because a proper subterm, namely ab, is an implicant. $ab\bar{c}$ is not a prime implicant either, because the proper subterm ab is an implicant, as is the proper subterm $b\bar{c}$.

We need one additional notion. An implicant is *redundant* if it can be omitted without changing the value of the predicate. As an example, the formula

$$f(a, b, c) = ab \vee ac \vee b\bar{c}$$

has three prime implicants, but the first one, ab, is redundant. A DNF representation is *minimal* if every implicant is prime and no implicant is redundant. Minimal DNF representations can be computed algebraically or by hand with Karnaugh maps, as discussed in a subsequent section.

With the above definitions, we can assume that we have a minimal DNF representation of a predicate. Given a minimal DNF representation for f, a *unique true point* with respect to the ith implicant is an assignment of truth values such that the ith implicant is true and all other implicants are false. It should be noted that if it is infeasible to make all of the "other" implicants false, then the implicant is redundant, violating our assumption that f is in minimal DNF form. The notion of unique true points allows a new criterion to be defined, *unique true point coverage (UTPC)*:

CRITERION **3.21 Unique True Point Coverage (UTPC):** *Given minimal DNF representations of a predicate f and its negation $\bar{f}$, TR contains a unique true point for each implicant in f and each implicant in $\bar{f}$.*

To return to our previous example:

$$f(a, b, c) = ab \vee b\bar{c}$$
$$\bar{f}(a, b, c) = \bar{b} \vee \bar{a}c$$

Both the f and $\bar{f}$ representations are minimal DNF representations. The following table shows the required assignments to the clauses a, b, and c to satisfy each implicant:

	a	b	c
ab	T	T	
$b\bar{c}$		T	F
$\bar{b}$		F	
$\bar{a}c$	F		T

The truth assignments to the remaining clauses must be such that no other implicants are true. For the first implicant, ab, c must be true or the second implicant will be true. For the second implicant, $b\bar{c}$, a must take the value false. For the third implicant, $\bar{b}$, there are three choices for unique true points: $\{FFF, TFF, TFT\}$. We have shown the first choice in the table below. Finally, for the fourth implicant, $\bar{a}c$, b must take the value true. The following table summarizes the discussion:

	a	b	c
ab	T	T	t
$b\bar{c}$	f	T	F
$\bar{b}$	f	F	f
$\bar{a}c$	F	t	T

Thus, the following test set satisfies UTPC:

$$T_2 = \{TTT,\ FTF,\ FFF,\ FTT\}$$

Although IC is relatively weak, UTPC is a fairly powerful coverage criterion. It is interesting to note that none of the active or inactive clause coverage criteria subsume UTPC. This can be seen by considering a simple counting argument. The DNF representation for a predicate with n clauses may have up to 2^{n-1} prime implicants in a minimal DNF representation. Hence, UTPC can require up to an exponential number of tests, which is more than the linear number $(n + 1)$ required by the clause coverage criteria. The potential exponential explosion does not automatically disqualify UTPC from practical consideration. For many predicates, UTPC generates a modest number of tests. One way of looking at this is that UTPC produces as many tests as the DNF representation of a predicate demands. In other words, more complex DNF representations demand more test cases.

UTPC does not subsume the active clause coverage criteria. A counterexample can be seen as follows. Consider the predicate

$$f(a, b, c) = ac \vee b\bar{c}$$
$$\bar{f}(a, b, c) = \bar{a}c \vee \bar{b}\bar{c}$$

A possible UTPC test set is

$$T_3 = \{TTT,\ TTF,\ FFT,\ FFF\}$$

The conditions under which c determines the value of f compute to $a \oplus b$. Notice that in all of the tests in T_3, a and b have the same value, and therefore $a \oplus b$ always evaluates to false. In other words, in this test set, c never determines the value of f. Hence, UTPC does not even subsume GACC, let alone CACC or RACC.

The literature contains a number of other DNF coverage criteria. The motivation for these coverage criteria is their ability to detect certain categories of faults. We already have a definition for unique true points. We need a corresponding definition for near false points to articulate these additional criteria. Given a DNF representation of a predicate f, a *near false point* for f with respect to clause c in implicant i is an assignment of truth values such that f is false, but if c is negated and all other clauses are left as is, i (and hence f) evaluates to true. For example, if f is

$$f(a, b, c, d) = ab \vee cd$$

then the near false points are $FTFF$, $FTFT$, $FTTF$ for clause a in the implicant ab, and $TFFF$, $TFFT$, $TFTF$ for clause b in the implicant ab. Corresponding unique true point and near false point pair coverage (CUTPNFP) is defined as follows:

> CRITERION **3.22 Corresponding Unique True Point and Near False Point Pair Coverage (CUTPNFP):** *Given a minimal DNF representation of a predicate f, for each clause c in each implicant i, TR contains a unique true point for i and a near false point for c in i such that the two points differ only in the truth value of c.*

By way of example, for

$$f(a, b, c, d) = ab \vee cd$$

if we consider clause a in the implicant ab, we can choose one of three unique true points, namely, $TTFF$, $TTFT$, $TTTF$, and pair each, respectively, with the corresponding near false points $FTFF$, $FTFT$, $FTTF$. So, for example, we could choose the first pair, $TTFF$, $FTFF$, to satisfy CUTPNFP with respect to clause a in implicant ab. Likewise, we could choose the pair $TTFF$, $TFFF$ to satisfy CUTPNFP with respect to clause b in implicant ab, the pair $FFTT$, $FFFT$ to satisfy CUTPNFP with respect to clause c in implicant cd, and the pair $FFTT$, $FFTF$ to satisfy CUTPNFP with respect to clause d in implicant cd, The resulting CUTPNFP set is

$$(TTFF, FFTT, FTFF, TFFF, FFFT, FFTF)$$

Note that the first two tests are unique true points, and the remaining four are corresponding near false points.

Table 3.8 defines a set of syntactic faults on predicates in DNF form.[3] Figure 3.6 gives a detection relationship between the types of faults in 3.8. Specifically, if a test set is guaranteed to detect a given type of fault, then the test set is also guaranteed to detect the types of faults "downstream" from that fault. Note that ENF is a particularly easy fault to catch – any test detects it.

As an example of relating the DNF coverage criteria to the fault classes, consider the UTPC criterion in the context of TOF faults. UTPC effectively detects TOF

Table 3.8. DNF fault classes

Fault	Description
Expression Negation Fault (ENF)	Expression wrongly implemented as its negation: e.g. $f = ab + c$ written as $f' = \overline{ab + c}$
Term Negation Fault (TNF)	A term is wrongly implemented as its negation: e.g. $f = ab + c$ written as $f' = \overline{ab} + c$
Term Omission Fault (TOF)	A term is wrongly omitted: e.g. $f = ab + c$ written as $f' = ab$
Literal Negation Fault (LNF)	A literal is wrongly implemented as its negation: e.g. $f = ab + c$ written as $f' = a\bar{b} + c$
Literal Reference Fault (LRF)	A literal is wrongly replaced by another literal: e.g. $f = ab + bcd$ written as $f' = ad + bcd$
Literal Omission Fault (LOF)	A literal is wrongly omitted: e.g. $f = ab + c$ written as $f' = a + c$
Literal Insertion Fault (LIF)	A literal is wrongly added to a term: e.g. $f = ab + c$ written as $f' = ab + \bar{b}c$
Operator Reference Fault (ORF+)	An 'Or' is wrongly replaced by 'And': e.g. $f = ab + c$ written as $f' = abc$
Operator Reference Fault (ORF*)	An 'And' is wrongly replaced by 'Or': e.g. $f = ab + c$ written as $f' = a + b + c$

faults. Note that UTPC demands a unique true point from the omitted implicant. Since the true point is unique, no other implicant will coincidentally yield the correct truth value on that test. Hence, the implementation will produce the negation of the desired truth value on the test, thereby revealing the TOF fault. Given the detection relationships in Figure 3.6, we can infer that UTPC also detects ORF+, LNF, TNF, and ENF faults. As another example, CUTPNFP effectively detects LOF faults. The reason why is that for every clause c in term i, CUTPNFP demands an unique true point and a near false point. These two tests differ only in the value of the clause c. Hence if the literal mentioning c is wrongly deleted in the implementation, both of these tests will produce the same truth value, thereby revealing the fault. Given the detection relationships in Figure 3.6, we can infer that CUTPNFP also detects ORF*, LNF, TNF, and ENF faults. In addition, although CUTPNFP does not subsume UTPC, CUTPNFP also detects the faults that UTPC detects.

Figure 3.6. Fault detection relationships.

CUTPNFP does not necessarily detect LIF faults, but it does subsume RACC. For other detection relations we refer the reader to the literature, where other, yet more powerful (and expensive!) criteria are defined, some of which do indeed detect LIF.

Karnaugh Maps

In this section, we review Karnaugh maps, which are exceedingly useful for producing DNF representations for predicates with a modest number of clauses.

A Karnaugh map is a tabular representation of a predicate with the special property that groupings of adjacent table entries correspond to simple DNF representations. Karnaugh maps are useful for predicates of up to four or five clauses; beyond that, they become cumbersome. A Karnaugh map for a predicate in four clauses is given below:

		ab			
		00	01	11	10
	00			1	
	01			1	
cd	11	1	1	1	1
	10			1	

Karnaugh map table for the predicate "ab ∨ cd".

For now, suppose that entries in the table are restricted to truth values. Truth values can be assigned in 2^{2^n} possible ways to the 2^n entries in a table for n clauses. So, the four clauses represented in the table above have 2^4 or 16 entries, and 2^{16} = 65, 536 possible functions. The reader will be relieved to know that we will not enumerate all of these in the text. Notice the labeling of truth values along the columns and rows. In particular, notice that any pair of adjacent cells differ in the truth value of exactly one clause. It might help to think of the edges of the Karnaugh map as being connected as well, so that the top and bottom rows are adjacent, as are the left and right columns (that is, a toroidal mapping from 2-space to 3-space).

The particular function represented in the Karnaugh map above can be spelled out in full:

$$ab\bar{c}\bar{d} \vee ab\bar{c}d \vee abcd \vee abc\bar{d} \vee \bar{a}\bar{b}cd \vee \bar{a}bcd \vee a\bar{b}cd$$

The expression simplifies to

$$ab \vee cd$$

The simplification can be read off the Karnaugh map by grouping together adjacent cells into rectangles of size 2^k for some $k > 0$ and forming rectangles of size 1 for cells with no adjacent cells. Overlaps among the groupings are fine. We give an example in three clauses to illustrate. Consider the following Karnaugh map:

		a, b			
		00	01	11	10
	0		1	1	
c	1	1		1	1

Four rectangles of size 2 can be extracted from this graph. They are the functions $b\bar{c}$, ab, ac, and $\bar{b}c$, and are represented by the following Karnaugh maps:

		a, b			
		00	01	11	10
	0		1	1	
c	1				

		a, b			
		00	01	11	10
	0			1	
c	1			1	

		a, b			
		00	01	11	10
	0				
c	1			1	1

		a, b			
		00	01	11	10
	0				
c	1	1			1

At first, the last of these might be a bit hard to see as a rectangle, but remember that the Karnaugh map is joined at the edges, left and right, as well as top and bottom. We could write the original function out as the disjunction of these four Karnaugh maps, each of which gives a prime implicant, but notice that the second, representing ab, is, in fact, redundant with the other three implicants, since all of its entries are covered by another of the Karnaugh maps. The resulting minimal DNF expression is

$$f = b\bar{c} \vee ac \vee \bar{b}c$$

One can also note that all of the entries of ac are covered by other Karnaugh maps, so ac is redundant with the remaining three implicants. So a different minimal DNF representation is

$$f = b\bar{c} \vee ab \vee \bar{b}c$$

Negations in DNF form are also easy to pull from a Karnaugh map. Consider again the function f given above. Negating f yields the Karnaugh map:

		a, b			
		00	01	11	10
	0	1			1
c	1		1		

Here, the three cells in the Karnaugh map can be covered with two rectangles, one of size 2, and the other of size 1. The resulting nonredundant, prime implicant formulation is

$$\bar{f} = \bar{b}\bar{c} \vee \bar{a}bc$$

Karnaugh maps are extremely convenient notations to derive test sets for many of the logic coverage criteria. For example, consider again the predicate $ab \vee cd$. Unique true points are simply true points covered by a single rectangle. Hence, of all the true points in $ab \vee cd$, all but $TTTT$ are unique true points. Near false points for any given true point are simply those false points that are immediately adjacent in the Karnaugh map. To determine UTPC tests, one simply takes the Karnaugh maps for f and $\bar{f}$ and, for each implicant, picks a cell covered only by that implicant. For CUTPNFP, one pairs up near false points with unique true points, being careful to obtain a pairing for each clause in f. Pairing of true points with near false points is also an easy way to develop RACC tests. Note that for RACC tests, it does not matter if the true points are unique or not.

EXERCISES

Section 3.6.

Use functions (1) through (4) to answer the following questions.

1. $f = ab\bar{c} + \bar{a}b\bar{c}$
2. $f = \bar{a}\bar{b}\bar{c}\bar{d} + abcd$
3. $f = ab + a\bar{b}c + \bar{a}\bar{b}c$
4. $f = \bar{a}\bar{c}\bar{d} + \bar{c}d + bcd$
   (a) Draw the Karnaugh maps for f and $\bar{f}$.
   (b) Find the nonredundant prime implicant representation for f and $\bar{f}$.
   (c) Give a test set that satisfies implicant coverage (IC) for f.
   (d) Give a test set that satisfies unique true point coverage (UTPC) for f.
   (e) Give a test set that satisfies Corresponding unique true point and near false point pair coverage (CUTPNFP) for f.

5 Use the following predicates to answer questions (a) through (e). In the questions, "simplest" means "fewest number of variable references."
   - $W = (B \wedge \neg C \wedge \neg D)$
   - $X = (B \wedge D) \vee (\neg B \neg D)$
   - $Y = (A \wedge B)$
   - $Z = (\neg B \wedge D)$

   (a) Draw the Karnaugh map for the predicates. Put AB on the top and CD on the side. Label each cell with W, X, Y, and/or Z as appropriate.
   (b) Write the simplest expression that describes all cells that have more than one definition.
   (c) Write the simplest expression that describes all cells that have no definitions.
   (d) Write the simplest expression that describes $X \vee Z$.
   (e) Give a test set for expression X that uses each prime implicant once.

6 Develop a counterexample to show that CUTPNFP does not subsume UPIC. Hint: You might wish to start with the expression given in the text, $f = ab + cd$.

3.7 BIBLIOGRAPHIC NOTES

The active clause criteria seem to have their beginnings in Myers' 1979 book [249]. A more accessible paper is by Zhu [367]. He defined decision and condition coverage, which Chilenski and Miller later used as a conceptual basis for MCDC [73, 305]. The definitions as originally given correspond to GACC in this book and did not address whether minor clauses had to have the same value for both values of the major clause. Chilenski also emphasized that the abbreviation should be "MCDC," not "MC/DC," and he has never put the '/' in the middle [72]. Most members of the aviation community interpreted MCDC to mean that the values of the minor clauses had to be the same, an interpretation that is called "unique-cause MCDC" [72]. Unique-cause MCDC corresponds to our RACC. More recently, the FAA has accepted the view that the minor clauses can differ, which is called "masking MCDC" [74]. Masking MCDC corresponds to our CACC. Our previous paper [17] clarified the definitions in the form used in this book and introduced the term "CACC."

The inactive clause criteria are adapted from the RC/DC method of Vilkomir and Bowen [332].

The result that the *internal variable* problem is formally undecidable is from Offutt's PhD dissertation [101, 262]. The problem is of primary importance in the automatic test data generation literature [36, 39, 101, 102, 150, 176, 179, 190, 191, 243, 295, 267].

Jasper et al. presented techniques for generating tests to satisfy MCDC [177]. They took the definition of MCDC from Chilenski and Miller's paper with the "default" interpretation that the minor clauses must be the same for both values of the major clauses. They went on to modify the interpretation so that if two clauses are coupled, which implies it is impossible to satisfy determination for both, the two clauses are allowed to have different values for the minor clauses. The fact that different values are allowed only when clauses are coupled puts their interpretation of MCDC between the RACC and CACC of this book.

Weyuker, Goradia, and Singh presented techniques for generating test data for software specifications that are limited to boolean variables [342]. The techniques were compared in terms of the ability of the resulting test cases to kill mutants (introduced in Chapter 5) [99, 101]. The results were that their technique, which is closely related to MCDC, performed better than any of the other techniques. Weyuker et al. incorporated syntax as well as meaning into their criteria. They presented a notion called *meaningful impact*, which is related to the notion of determination, but which has a syntactic basis rather than a semantic one.

Kuhn investigated methods for generating tests to satisfy various decision-based criteria, including MCDC tests [194]. He used the definition from Chilenski and Miller [73, 305], and proposed the boolean derivative to satisfy MCDC. In effect, this interpreted MCDC in such a way to match CACC.

Dupuy and Leveson's 2000 paper evaluated MCDC experimentally [108]. They presented results from an empirical study that compared pure functional testing with functional testing augmented by MCDC. The experiment was performed during the testing of the attitude control software for the HETE-2 (High Energy Transient Explorer) scientific satellite. The definition of MCDC from their paper is the traditional definition given in the FAA report and Chilenski and Miller's paper: "Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to affect that decision outcome independently. A condition is shown to affect a decision's outcome independently by varying just that decision while holding fixed all other possible conditions."

Note the misstatement in last line: "varying just that decision" should be "varying just that condition." This does not say that the decision has a different value when the condition's value changes. "Holding fixed" can be assumed to imply that the minor clauses cannot change with different values for the major clause (that is, RACC, not CACC).

The full predicate method of Offutt, Liu, Abdurazik and Ammann [272] explicitly relaxes the requirement that the major clauses have the same value as the predicate. This is equivalent to CACC and almost the same as masking MCDC.

Jones and Harrold have developed a method for reducing the regression tests that were developed to satisfy MCDC [180]. They defined MCDC as follows: "MC/DC is a stricter form of decision (or branch) coverage.... MC/DC requires that each condition in a decision be shown by execution to independently affect the outcome of the decision." This is taken directly from Chilenski and Miller's original paper, and their interpretation of the definition is the same as CACC.

SCR was first discussed by Henninger [157] and its use in model checking and testing was introduced by Atlee [20, 21].

The method of determining p_c given in this book uses the boolean derivative developed by Akers [6]. Both Chilenski and Richey [74] and Kuhn [194] applied Akers's derivative to exactly the problem given in this chapter. The other methods are the pairs table method of Chilenski and Miller and the tree method, independently discovered by Chilenski and Richey [74] and Offutt et al. [272]. The tree method implements the boolean derivative method in a procedural way.

Ordered binary decision diagrams (OBDDs) offer another way of determining p_c. In particular, consider any OBDD in which clause c is ordered last. Then any path through the OBDD that reaches a node labeled c (there will be exactly zero, one, or two such nodes) is, in fact, an assignment of values to the other variables so that c determines p. Continuing the path on to the constants T and F yields a pair of tests satisfying $RACC$ with respect to c. Selecting two different paths that reach the same node labeled c, and then extending each so that one reaches T and the other reaches F yields a pair of tests that satisfy $CACC$, but not $RACC$, with respect to c. Finally, if two nodes are labeled c, then it is possible to satisfy $GACC$ but not $CACC$ with respect to c: Select paths to each of the two nodes labeled c, extend one path by choosing c true, and the other by choosing c false. Both paths will necessarily end up in the same node, namely, either T or F. ICC tests with respect to c can be derived by considering paths to T and F in the OBDD where the paths

do not include variable c. The attractive aspect of using OBDDs to derive ACC or ICC tests is that a variety of existing tools can handle a relatively large number of clauses. The unattractive aspect is that for a predicate with N clauses, N different OBDDs for a given function are required, since the clause being attended to needs to be the last in the ordering. To the knowledge of the authors, the use of OBDDs to derive ACC or ICC tests does not appear in the literature.

Beizer's book [29] includes a chapter on DNF testing, including a variant of IC coverage for f, but not $\bar{f}$, and an extensive development of Karnaugh maps. We do not address criteria defined predicates expressed in conjunctive normal form (CNF). The reason is that every DNF coverage criteria has a dual in CNF. Kuhn [194] developed the first fault detection relations; this work was greatly expanded by Yu, Lau, and Chen, who developed much of the key material relating DNF coverage criteria to fault detecting ability. Two good papers to begin study of this topic are by Chen and Lau [63], which develops a variety of coverage criteria, including CUTPNFP, and Lau and Yu [202], which is the source for the fault class hierarchy shown in Figure 3.6. In personal communications, Greg Williams and Gary Kaminski provided the authors with valuable assistance in organizing and expanding the DNF fault detection material. Greg Williams also developed the counterexample to show that UTPC and the ACC criteria do not share a subsumption relation.

NOTES

1 In practice, this "thought" turned out to be the collective effort of many researchers, who published dozens of papers over a period of several decades.

2 Old hands at testing may recognize and even be tired of the triangle example. It is used as a teaching tool for the same reasons it has staying power in the literature: it's a familiar problem; the control structure is interesting enough to illustrate most issues; and it does not use language features that make this analysis really hard, such as loops and indirect references. This version of `TriTyp` is a bit overly complicated, but that does help illustrate the concepts.

3 The notion of mutation operators developed in the Chapter 5 is closely related to the notion of fault classes presented here.

4

Input Space Partitioning

In a very fundamental way, all testing is about choosing elements from the input space of the software being tested. The criteria presented previously can be viewed as defining ways to divide the input space according to the test requirements. The assumption is that any collection of values that satisfies the same test requirement will be "just as good." Input space partitioning takes that view in a much more direct way. The input *domain* is defined in terms of the possible values that the input parameters can have. The input parameters can be method parameters and global variables, objects representing current state, or user-level inputs to a program, depending on what kind of software artifact is being analyzed. The input domain is then partitioned into regions that are assumed to contain equally useful values from a testing perspective, and values are selected from each region.

This way of testing has several advantages. It is fairly easy to get started because it can be applied with no automation and very little training. The tester does not need to understand the implementation; everything is based on a description of the inputs. It is also simple to "tune" the technique to get more or fewer tests.

Consider an abstract partition q over some domain D. The partition q defines a set of equivalence classes, which we simply call *blocks*, B_q.[1] The blocks are pairwise disjoint, that is

$$b_i \cap b_j = \emptyset, i \neq j; b_i, b_j \in B_q$$

and together the blocks cover the domain D, that is

$$\bigcup_{b \in B_q} b = D$$

This is illustrated in Figure 4.1. The input domain D is partitioned into three blocks, b_1, b_2, and b_3. The partition defines the values contained in each block and is usually designed from knowledge of what the software is supposed to do.

The idea in partition coverage is that any test in a block is as good as any other for testing. Several partitions are sometimes considered together, which, if not done carefully, leads to a combinatorial explosion of test cases.

Figure 4.1. Partitioning of input domain *D* into three blocks.

A common way to apply input space partitioning is to start by considering the domain of each parameter separately, partitioning each domain's possible values into blocks, and then combining the variables for each parameter. Sometimes the parameters are considered completely independently, and sometimes they are considered in conjunction, usually by taking the semantics of the program into account. This process is called *input domain modeling* and the next section gives more details.

Each partition is usually based on some *characteristic C* of the program, the program's inputs, or the program's environment. Some possible characteristic examples are:

- Input X is null
- Order of file F (sorted, inverse sorted, arbitrary)
- Min separation distance of two aircraft

Each characteristic C allows the tester to define a partition. Formally, a partition *must* satisfy two properties:

1. The partition must cover the entire domain (completeness)
2. The blocks must not overlap (disjoint)

As an example, consider the characteristic "order of file F" mentioned above. This could be used to create the following (defective) partitioning:

- Order of file F
  - b_1 = Sorted in ascending order
  - b_2 = Sorted in descending order
  - b_3 = Arbitrary order

However, this is **not** a valid partitioning. Specifically, if the file is of length 0 or 1, then the file will belong in all three blocks. That is, the blocks are not disjoint. The easiest strategy to address this problem is to make sure that each characteristic addresses only one property. The problem above is that the notions of being sorted into ascending order and being sorted into descending order are lumped into the same characteristic. Splitting into two characteristics, namely sorted ascending and sorted descending, solves the problem. The result is the following (valid) partitioning of two characteristics.

- File F sorted ascending
  - b_1 = True
  - b_2 = False

- File F sorted descending
  - b_1 = True
  - b_2 = False

With these blocks, files of length 0 or 1 are in the True block for both characteristics.

The completeness and disjointness properties are formalized for pretty pragmatic reasons, and not just to be mathematically fashionable. Partitions that are not complete or disjoint probably reflect a lack of clarity in the rationale for the partition. In particular, if a partition actually encodes two or three rationales, the partition is likely to be quite messy, and it is also likely to violate either the completeness or the disjointness property (or both!). Identifying and correcting completeness or disjointness errors typically results in esthetically more pleasing partitions. Further, formally objectionable "partitions" cause unnecessary problems when generating tests, as discussed below. The rest of this chapter assumes that the partitions are both complete and disjoint.

4.1 INPUT DOMAIN MODELING

The first step in input domain modeling is identification of testable functions. Consider the `TriTyp` program from Chapter 3. `TriTyp` clearly has only one testable function with three parameters. The situation is more complex for Java class APIs. Each public method is typically a testable function that should be tested individually. However, the characteristics are often the same for several methods, so it helps to develop a common set of characteristics for the entire class and then develop specific tests for each method. Finally, large systems are certainly amenable to the input space partition approach, and such systems often supply complex functionality. Tools like UML use cases can be used to identify testable functions. Each use case is associated with a specific intended functionality of the system, so it is very likely that the use case designers have useful characteristics in mind that are relevant to developing test cases. For example, a "withdrawal" use case for an ATM identifies "withdrawing cash" as a testable function. Further, it suggests useful categories such as "Is Card Valid?" and "Relation of Withdrawal Policy to Withdrawal Request."

The second step is to identify all of the parameters that can affect the behavior of a given testable function. This step isn't particularly creative, but it is important to carry it out completely. In the simple case of testing a stateless method, the parameters are simply the formal parameters to the method. If the method has state, which is common in many object-oriented classes, then the state must be included as a parameter. For example, the `insert(Comparable obj)` method for a binary tree class behaves differently depending on whether or not `obj` is already in the tree. Hence, the current state of the tree needs to be explicitly identified as a parameter to the `insert()` method. In a slightly more complex example, a method `find(String str)` that finds the location of `str` in a file depends, obviously, on the particular file being searched. Hence, the test engineer explicitly identifies the file as a parameter to the `find()` method. Together, all of the parameters form the *input domain* of the function under test.

The third step, and the key creative engineering step, is modeling the input domain articulated in the prior step. An *input domain model* (*IDM*) represents the input space of the system under test in an abstract way. A test engineer describes

the structure of the input domain in terms of input *characteristics*. The test engineer creates a *partition* for each characteristic. The partition is a set of *blocks*, each of which contains a set of *values*. From the perspective of that particular characteristic, the values in each block are considered equivalent.

A test input is a tuple of values, one for each parameter. By definition, the test input belongs to exactly one block from each characteristic. Thus, if we have even a modest number of characteristics, the number of possible combinations may be infeasible. In particular, adding another characteristic with n blocks increases the number of combinations by a factor of n. Hence, controlling the total number of combinations is a key feature of any practical approach to input domain testing. In our view, this is the job of the coverage criteria, which we address in Section 4.2.

Different testers will come up with different models, depending on creativity and experience. These differences create a potential for variance in the quality of the resulting tests. The structured method to support input domain modeling presented in this chapter can decrease this variance and increase the overall quality of the IDM.

Once the IDM is built and values are identified, some combinations of the values may be invalid. The IDM must include information to help the tester identify and avoid or remove invalid sub-combinations. The model needs a way to represent these restrictions. Constraints are discussed further in Section 4.3.

The next section provides two different approaches to input domain modeling. The *interface-based* approach develops characteristics directly from input parameters to the program under test. The *functionality-based* approach develops characteristics from a functional or behavioral view of the program under test. The tester must choose which approach to use. Once the IDM is developed, several coverage criteria are available to decide which combinations of values to use to test the software. These are discussed in Section 4.2.

4.1.1 Interface-Based Input Domain Modeling

The interface-based approach considers each particular parameter in isolation. This approach is almost mechanical to follow, but the resulting tests are surprisingly good.

An obvious strength of using the interface-based approach is that it is easy to identify characteristics. The fact that each characteristic limits itself to a single parameter also makes it easy to translate the abstract tests into executable test cases.

A weakness of this approach is that not all the information available to the test engineer will be reflected in the interface domain model. This means that the IDM may be incomplete and hence additional characteristics are needed.

Another weakness is that some parts of the functionality may depend on combinations of specific values of several interface parameters. In the interface-based approach each parameter is analyzed in isolation with the effect that important sub-combinations may be missed.

Consider the `TriTyp` program from Chapter 3. It has three integer parameters that represent the lengths of three sides of a triangle. In an interface-based IDM, `Side1` will have a number of characteristics, as will `Side2` and `Side3`. Since the three variables are all of the same type, the interface-based characteristics for each will

likely be identical. For example, since Side1 is an integer, and zero is often a special value for integers, Relation of Side1 to zero is a reasonable interface-based characteristic.

4.1.2 Functionality-Based Input Domain Modeling

The idea of the functionality-based approach is to identify characteristics that correspond to the intended functionality of the system under test rather than using the actual interface. This allows the tester to incorporate some semantics or domain knowledge into the IDM.

Some members of the community believe that a functionality-based approach yields better test cases than the interface-based approach because the input domain models include more semantic information. Transferring more semantic information from the specification to the IDM makes it more likely to generate expected results for the test cases, an important goal.

Another important strength of the functionality-based approach is that the requirements are available before the software is implemented. This means that input domain modeling and test case generation can start early in development.

In the functionality-based approach, identifying characteristics and values may be far from trivial. If the system is large and complex, or the specifications are informal and incomplete, it can be very hard to design reasonable characteristics. The next section gives practical suggestions for designing characteristics.

The functionality-based approach also makes it harder to generate tests. The characteristics of the IDM often do not map to single parameters of the software interface. Translating the values into executable test cases is harder because constraints of a single IDM characteristic may affect multiple parameters in the interface.

Returning to the TriTyp program from Chapter 3, a functionality-based approach will recognize that instead of simply three integers, the input to the method is a triangle. This leads to the characteristic of a triangle, which can be partitioned into different types of triangles (as discussed below).

4.1.3 Identifying Characteristics

Identifying characteristics in an interface-based approach is simple. There is a mechanical translation from the parameters to characteristics. Developing a functionality-based IDM is more challenging.

Preconditions are excellent sources for functionality-based characteristics. They may be explicit or encoded in the software as exceptional behavior. Preconditions explicitly separate defined (or normal) behavior from undefined (or exceptional) behavior. For example, if a method choose() is supposed to select a value, it needs a precondition that a value must be available to select. A characteristic may be whether the value is available or not.

Postconditions are also good sources for characteristics. In the case of TriTyp, the different kinds of triangles are based on the postcondition of the method.

The test engineer should also look for other relationships between variables. These may be explicit or implicit. For example, a curious test engineer given a

method m() with two object parameters x and y might wonder what happens if x and y point to the same object (aliasing), or to logically equal objects.

Another possible idea is to check for missing factors, that is, factors that may impact the execution but do not have an associated IDM parameter.

It is usually better to have many characteristics with few blocks than the reverse. It is also true that characteristics with small numbers of blocks are more likely to satisfy the disjointness and completeness properties.

Generally, it is preferable for the test engineer to use specifications or other documentation instead of program code to develop characteristics. The idea is that the tester should apply input space partitioning by using *domain knowledge* about the problem, not the implementation. However, in practice, the code may be all that is available. Overall, the more semantic information the test engineer can incorporate into characteristics, the better the resulting test set is likely to be.

The two approaches generally result in different IDM characteristics. The following method illustrates this difference:

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//    else returns true if element is in the list, false otherwise
```

If the interface-based approach is used, the IDM will have characteristics for list and characteristics for element. For example, here are two interface-based characteristics for list, including blocks and values, which are discussed in detail in the next section:

- list is null
  - b_1 = True
  - b_2 = False
- list is empty
  - b_1 = True
  - b_2 = False

The functionality-based approach results in more complex IDM characteristics. As mentioned earlier, the functionality-based approach requires more thinking on the part of the test engineer, but can result in better tests. Two possibilities for the example are listed below, again including blocks and values.

- number of occurrences of element in list
  - $b_1 = 0$
  - $b_2 = 1$
  - b_3 = More than 1
- element occurs first in list
  - b_1 = True
  - b_2 = False

4.1.4 Choosing Blocks and Values

After choosing characteristics, the test engineer partitions the domains of the characteristics into sets of values called *blocks*. A key issue in any partition approach is how partitions should be identified and how representative values should be selected from each block. This is another creative design step that allows the tester to tune the test process. More blocks will result in more tests, requiring more resources but possibly finding more faults. Fewer blocks will result in fewer tests, saving resources but possibly reducing test effectiveness. Several general strategies for identifying values are as follows:

- **Valid values**: Include at least one group of valid values.
- **Sub-partition**: A range of valid values can often be partitioned into sub-partitions, such that each sub-partition exercises a somewhat different part of the functionality.
- **Boundaries**: Values at or close to boundaries often cause problems.
- **Normal use**: If the operational profile focuses heavily on "normal use," the failure rate depends on values that are not boundary conditions.
- **Invalid values**: Include at least one group of invalid values.
- **Balance**: From a cost perspective, it may be cheap or even free to add more blocks to characteristics that have fewer blocks. In Section 4.2, we will see that the number of tests sometimes depends on the characteristic with the maximum number of blocks.
- **Missing partitions**: Check that the union of all blocks of a characteristic completely covers the input space of that characteristic.
- **Overlapping partitions**: Check that no value belongs to more than one block.

Special values can often be used. Consider a Java reference variable; null is typically a special case that needs to be treated differently from non null values. If the reference is to a container structure such as a Set or List, then whether the container is empty or not is often a useful characteristic.

Consider the TriTyp program from Chapter 3. It has three integer parameters that represent the lengths of three sides of a triangle. One common partitioning for an integer variable considers the relation of the variable's value to some special value in the testable function's domain, such as zero.

Table 4.1 shows a partitioning for the interface-based IDM for the TriTyp program. It has three characteristics, q_1, q_2, and q_3.

The first row in the table should be read as "Block $q_1.b_1$ is that Side 1 is greater than zero," "Block $q_1.b_2$ is that Side 1 is equal to zero," and "Block $q_1.b_3$ is that Side 1 is less than zero."

Table 4.1. First partitioning of TriTyp's inputs (interface-based)

Partition	b_1	b_2	b_3
q_1 = "Relation of Side 1 to 0"	*greater than 0*	*equal to 0*	*less than 0*
q_2 = "Relation of Side 2 to 0"	*greater than 0*	*equal to 0*	*less than 0*
q_3 = "Relation of Side 3 to 0"	*greater than 0*	*equal to 0*	*less than 0*

Table 4.2. Second partitioning of TriTyp's inputs (interface-based).

Partition	b_1	b_2	b_3	b_4
q_1 = "Length of Side 1"	*greater than 1*	*equal to 1*	*equal to 0*	*less than 0*
q_2 = "Length of Side 2"	*greater than 1*	*equal to 1*	*equal to 0*	*less than 0*
q_3 = "Length of Side 3"	*greater than 1*	*equal to 1*	*equal to 0*	*less than 0*

Consider the partition q_1 for Side 1. If one value is chosen from each block, the result is three tests. For example, we might choose Side 1 to have the value 7 in test 1, 0 in test 2, and −3 in test 3. Of course, we also need values for Side 2 and Side 3 of the triangle to complete the test case values. Notice that some of the blocks represent valid triangles and some represent invalid triangles. For example, no valid triangle can have a side of negative length.

It is easy to refine this categorization to get more fine grained testing if the budget allows. For example, more blocks can be created by separating inputs with value 1. This decision leads to a partitioning with four blocks, as shown in Table 4.2.

Notice that if the value for Side 1 were floating point rather than integer, the second categorization would **not** yield valid partitions. None of the blocks would include values between 0 and 1 (noninclusive), so the blocks would not cover the domain (not be complete). However, the domain D contains integers so the partitions are valid.

While partitioning, it is often useful for the tester to identify candidate values for each block to be used in testing. The reason to identify values now is that choosing specific values can help the test engineer think more concretely about the predicates that describe each block. While these values may not prove sufficient when refining test requirements to test cases, they do form a good starting point. Table 4.3 shows values that can satisfy the second partitioning.

The above partitioning is interface based and only uses syntactic information about the program (it has three integer inputs). A functionality-based approach can use the semantic information of the traditional geometric classification of triangles, as shown in Table 4.4.

Of course, the tester has to know what makes a triangle scalene, equilateral, isosceles, and invalid to choose possible values (this may be simple middle school geometry, but many of us have probably forgotten). An equilateral triangle is one in which all sides are the same length. An isosceles triangle is one in which at least two sides are the same length. A scalene triangle is any other valid triangle. This brings up a subtle problem, Table 4.4 does **not** form a valid partitioning. An equilateral triangle is also isosceles, thus we must first correct the partitions, as shown in Table 4.5.

Table 4.3. Possible values for blocks in the second partitioning in Table 4.2

Param	b_1	b_2	b_3	b_4
Side 1	2	1	0	−1
Side 2	2	1	0	−1
Side 3	2	1	0	−1

Table 4.4. Geometric partitioning of TriTyp's inputs (functionality-based)

Partition	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	*scalene*	*isosceles*	*equilateral*	*invalid*

Now values for Table 4.5 can be chosen as shown in Table 4.6. The triplets represent the three sides of the triangle.

A different approach to the equilateral/isosceles problem above is to break the characteristic Geometric Partitioning into four separate characteristics, namely Scalene, Isosceles, Equilateral, and Valid. The partition for each of these characteristics is boolean, and the fact that choosing Equilateral = true also means choosing Isosceles = true is then simply a constraint. Such an approach is highly recommended, and it invariably satisfies the disjointness and completeness properties.

4.1.5 Using More than One Input Domain Model

For a complex program it might be better to have several small IDMs than one large. This approach allows for a divide-and-conquer strategy when modeling characteristics and blocks. Another advantage with multiple IDMs for the same software is that it allows varying levels of coverage.

For instance, one IDM may contain only valid values and another IDM may contain invalid values to focus on error handling. The valid value IDM may be covered using a higher level of coverage. The invalid value IDM may use a lower level of coverage.

Multiple IDMs may be overlapping as long as the test cases generated make sense. However, overlapping IDMs are likely to have more constraints.

4.1.6 Checking the Input Domain Model

It is important to check the input domain model. In terms of characteristics, the test engineer should ask whether there is any information about how the function behaves that is not incorporated in some characteristics. This is necessarily an informal process.

The tester should also explicitly check each characteristic for the completeness and disjointness properties. The purpose of this check is to make sure that, for each characteristic, not only do the blocks cover the complete input space, but selecting a particular block implies excluding all other blocks in that characteristic.

If multiple IDMs are used, completeness should be relative to the portion of the input domain that is modeled in each IDM. When the tester is satisfied with the

Table 4.5. Correct geometric partitioning of TriTyp's inputs (functionality-based)

Partition	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	*scalene*	*isosceles, not equilateral*	*equilateral*	*invalid*

Table 4.6. Possible values for blocks in geometric partitioning in Table 4.5.

Param	b_1	b_2	b_3	b_4
Triangle	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

characteristics and their blocks, it is time to choose which combinations of values to test with and identify constraints among the blocks.

EXERCISES

Section 4.1.

1. Answer the following questions for the method search() below:

```
public static int search (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//    else if element is in the list, return an index
//    of element in the list; else return -1
//    for example, search ([3,3,1], 3) = either 0 or 1
//        search ([1,7,5], 2) = -1
```

   Base your answer on the following characteristic partitioning:

```
Characteristic: Location of element in list
    Block 1: element is first entry in list
    Block 2: element is last entry in list
    Block 3: element is in some position other than first or last
```

   (a) "Location of element in list" fails the disjointness property. Give an example that illustrates this.
   (b) "Location of element in list" fails the completeness property. Give an example that illustrates this.
   (c) Supply one or more new partitions that capture the intent of "Location of e in list" but do not suffer from completeness or disjointness problems.
2. Derive input space partitioning tests for the **GenericStack** class with the following method signatures:
   - public GenericStack ();
   - public void Push (Object X);
   - public Object Pop ();
   - public boolean IsEmt ();

   Assume the usual semantics for the stack. Try to keep your partitioning simple, choose a small number of partitions and blocks.
   (a) Define characteristics of inputs
   (b) Partition the characteristics into blocks
   (c) Define values for the blocks

4.2 COMBINATION STRATEGIES CRITERIA

The above description ignores an important question: "How should we consider multiple partitions at the same time?" This is the same as asking "What combination of blocks should we choose values from?" For example, we might wish to require a test case that satisfies block 1 from q_2 and block 3 from q_3. The most obvious choice is to choose all combinations. However, just like Combinatorial Coverage from previous chapters, using all combinations will be impractical when more than 2 or 3 partitions are defined.

> CRITERION **4.23 All Combinations Coverage (ACoC):** *All combinations of blocks from all characteristics must be used.*

For example, if we have three partitions with blocks [A, B], [1, 2, 3], and [x, y], then ACoC will need the following twelve tests:

(A, 1, x)	(B, 1, x)
(A, 1, y)	(B, 1, y)
(A, 2, x)	(B, 2, x)
(A, 2, y)	(B, 2, y)
(A, 3, x)	(B, 3, x)
(A, 3, y)	(B, 3, y)

A test suite that satisfies ACoC will have a unique test for each combination of blocks for each partition. The number of tests will be the product of the number of blocks for each partition: $\prod_{i=1}^{Q}(B_i)$.

If we use a four block partition similar to q_2 for each of the three sides of the triangle, ACoC requires $4 * 4 * 4 = 64$ tests.

This is almost certainly more testing than is necessary, and will usually be economically impractical as well. Thus, as with paths and truth tables before, we must use some sort of coverage criterion to choose which combinations of blocks to pick values from.

The first, fundamental assumption is that different choices of values from the same block are equivalent from a testing perspective. That is, we need to take only one value from each block. Several *combination strategies* exist, which result in a collection of useful criteria. These combination strategies are illustrated with the `TriTyp` example, using the second categorization given in Table 4.2 and the values from Table 4.3.

The first combination strategy criterion is fairly straightforward and simply requires that we try each choice at least once.

> CRITERION **4.24 Each Choice Coverage (ECC):** *One value from each block for each characteristic must be used in at least one test case.*

Given the above example of three partitions with blocks [A, B], [1, 2, 3], and [x, y], ECC can be satisfied in many ways, including the three tests (A, 1, x), (B, 2, y), and (A, 3, x).

Assume the program under test has Q parameters $q_1, q_2, \ldots, q_Q$, and each parameter q_i has B_i blocks. Then a test suite that satisfies ECC will have at least $Max_{i=1}^{Q} B_i$

values. The maximum number of blocks for the partitions for TriTyp is four, thus ECC requires at least four tests.

This criterion can be satisfied by choosing the tests {(2, 2, 2), (1, 1, 1), (0, 0, 0), (−1, −1, −1)} from Table 4.3. It does not take much thought to conclude that these are not very effective tests for this program. ECC leaves a lot of flexibility to the tester in terms of how to combine the test values, so it can be called a relatively "weak" criterion.

The weakness of ECC can be expressed as not requiring values to be combined with other values. A natural next step is to require explicit combinations of values, called *pair-wise*.

> CRITERION **4.25 Pair-Wise Coverage (PWC):** *A value from each block for each characteristic must be combined with a value from every block for each other characteristic.*

Given the above example of three partitions with blocks [A, B], [1, 2, 3], and [x, y], then PWC will need sixteen tests to cover the following combinations:

(A, 1)	(B, 1)	(1, x)
(A, 2)	(B, 2)	(1, y)
(A, 3)	(B, 3)	(2, x)
(A, x)	(B, x)	(2, y)
(A, y)	(B, y)	(3, x)
		(3, y)

PWC allows the same test case to cover more than one unique pair of values. So the above combinations can be combined in several ways, including:

(A, 1, x)	(B, 1, y)
(A, 2, x)	(B, 2, y)
(A, 3, x)	(B, 3, y)
(A, –, y)	(B, –, x)

The tests with '–' mean that any block can be used.

A test suite that satisfies PWC will pair each value with each other value or have at least $(Max_{i=1}^{Q} B_i)^2$ values. Each characteristic in TriTyp (Table 4.3) has four blocks; so at least 16 tests are required.

Several algorithms to satisfy PWC have been published and appropriate references are provided in the bibliography section of the chapter.

A natural extension to PWC is to require t values instead of pairs.

> CRITERION **4.26 T-Wise Coverage (TWC):** *A value from each block for each group of t characteristics must be combined.*

If the value for T is chosen to be the number of partitions, Q, then TWC is equivalent to all combinations. A test suite that satisfies TWC will have at least $(Max_{i=1}^{q} B_i)^t$ values. TWC is expensive in terms of the number of test cases, and experience suggests going beyond pair-wise (that is, $t = 2$) does not help much.

Both PWC and TWC combine values "blindly," without regard for which values are being combined. The next criterion strengthens ECC in a different way by bringing in a small but crucial piece of domain knowledge of the program; asking what is the most "important" block for each partition. This block is called the *base choice*.

> CRITERION **4.27 Base Choice Coverage (BCC):** *A base choice block is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.*

Given the above example of three partitions with blocks [A, B], [1, 2, 3], and [x, y], suppose base choice blocks are 'A', '1' and 'x'. Then the base choice test is (A, 1, x), and the following additional tests would need to be used:

(B, 1, x)
(A, 2, x)
(A, 3, x)
(A, 1, y)

A test suite that satisfies BCC will have one base test, plus one test for each remaining block for each partition. This is a total of $1 + \sum_{i=1}^{Q}(B_i - 1)$. Each parameter for TriTyp has four blocks, thus BCC requires $1 + 3 + 3 + 3$ tests.

The base choice can be the simplest, the smallest, the first in some ordering, or the most likely from an end-user point of view. Combining more than one invalid value is usually not useful because the software often recognizes one value and negative effects of the others are masked. Which blocks are chosen for the base choices becomes a crucial step in test design that can greatly impact the resulting test. It is important that the tester document the strategy that was used so that further testing can reevaluate those decisions.

Following the strategy of choosing the most likely block for TriTyp, we chose "greater than 1" from Table 4.2 as the base choice block. Using the values from Table 4.3 gives the base test as (2, 2, 2). The remaining tests are created by varying each one of these in turn: {(2, 2, 1), (2, 2, 0), (2, 2, −1), (2, 1, 2), (2, 0, 2), (2, −1, 2), (1, 2, 2), (0, 2, 2), (−1, 2, 2) }.

Sometimes the tester may have trouble choosing a single base choice and may decide that multiple base choices are needed. This is formulated as follows:

> CRITERION **4.28 Multiple Base Choices (MBCC):** *At least one, and possibly more, base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic at least once. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choice in each other characteristic.*

Assuming m_i base choices for each characteristic and a total of M base tests, MBCC requires $M + \sum_{i=1}^{Q}(M * (B_i - m_i))$ tests.

For example, we may choose to include two base choices for side 1 in TriTyp, "greater than 1" and "equal to 1." This would result in the two base tests (2, 2, 2) and (1, 2, 2). The formula above is thus evaluated with $M = 2$, $m_1 = 2$, and

Figure 4.2. Subsumption relations among input space partitioning criteria.

$m_i = 1 \ \forall \ i, 1 < i \leq 3$. That is, $2 + (2*(4-2)) + (2*(4-1)) + (2*(4-1)) = 18$. The remaining tests are created by varying each one of these in turn. The MBCC criterion sometimes results in duplicate tests. For example, $(0, 2, 2)$ and $(-1, 2, 2)$ both appear twice for TriTyp. Duplicate test cases should, of course, be eliminated (which also makes the formula for the number of tests an upper bound).

Figure 4.2 shows the subsumption relationships among the input space partitioning combination strategy criteria.

EXERCISES

Section 4.2.

1. Enumerate all 64 tests to satisfy the All Combinations (ACoC) criterion for the second categorization of TriTyp's inputs in Table 4.2. Use the values in Table 4.3.
2. Enumerate all 16 tests to satisfy the pair-wise (PWC) criterion for the second categorization of TriTyp's inputs in Table 4.2. Use the values in Table 4.3.
3. Enumerate all 16 tests to satisfy the multiple base choice (MBCC) criterion for the second categorization of TriTyp's inputs in Table 4.2. Use the values in Table 4.3.
4. Answer the following questions for the method intersection() below:

```
public Set intersection (Set s1, Set s2)
 // Effects:  If s1 or s2 are null throw NullPointerException
 //    else return a (non null) Set equal to the intersection
 //    of Sets s1 and s2
 //    A null argument is treated as an empty set.
```

```
Characteristic: Type of s1
  - s1 = null
  - s1 = {}
  - s1 has at least one element

Characteristic: Relation between s1 and s2
  - s1 and s2 represent the same set
  - s1 is a subset of s2
  - s2 is a subset of s1
  - s1 and s2 do not have any elements in common
```

(a) Does the partition "Type of s1" satisfy the completeness property? If not, give a value for $s1$ that does not fit in any block.
(b) Does the partition "Type of s1" satisfy the disjointness property? If not, give a value for $s1$ that fits in more than one block.
(c) Does the partition "Relation between s1 and s2" satisfy the completeness property? If not, give a pair of values for $s1$ and $s2$ that does not fit in any block.
(d) Does the partition "Relation between s1 and s2" satisfy the disjointness property? If not, give a pair of values for $s1$ and $s2$ that fits in more than one block.
(e) If the "base choice" criterion were applied to the two partitions (exactly as written), how many test requirements would result?

5. Derive input space partitioning tests for the **BoundedQueue** class with the following signature:

   - `public BoundedQueue (int capacity);`
   - `public void Enqueue (Object X);`
   - `public Object Dequeue ();`
   - `public boolean IsEmpty ();`
   - `public boolean IsFull ();`

   Assume the usual semantics for a queue with a fixed, maximal capacity. Try to keep your partitioning simple–choose a small number of partitions and blocks.
   (a) Identify all of the variables. Don't forget the state variables.
   (b) Identify several characteristics that suggest partitions.
   (c) Identify the blocks in the partition for each characteristic. Designate one block in each partition as the "Base" block.
   (d) Define values for the blocks.
   (e) Define a test set that satisfies base choice coverage (BCC).
6. Develop a set of characteristics and accompanying partitions for the pattern checking procedure (the method *pat()* in Figure 2.21 in Chapter 2).
   (a) Develop tests to satisfy the base choice criterion. Your tests should have both inputs and expected outputs.
   (b) Analyze your tests with respect to the data flow test sets developed in Chapter 2. How well does input space partitioning do?

Table 4.7. Examples of invalid block combinations

	Blocks			
Characteristics	**1**	**2**	**3**	**4**
A: length and contents	one element	more than one, unsorted	more than one, sorted	more than one, all identical
B: match	element not found	element found once	element found more than once	–

Invalid combinations: (A1, B3), (A4, B2)

4.3 CONSTRAINTS AMONG PARTITIONS

A subtle point about input space partitioning is that some combinations of blocks are infeasible. This must be documented in the IDM. For example, Table 4.7 shows an example based on the previously described `boolean findElement (list, element)` method. An IDM with two parameters *A*, that has four partitions, and *B*, that has three partitions, has been designed. Two of the partition combinations do not make sense and are thus invalid. In this example, these are represented as a list of invalid pairs of parameter partitions. In the general case other representations can be used, for example, a set of inequalities.

Constraints are relations between blocks from different characteristics. Two kinds of constraints appear. One kind says that a block from one characteristic cannot be combined with a block from another characteristic. The "less than zero" and "scalene" problem is an example of this kind of constraint. The other kind is the inverse; a block from one characteristic **must be** combined with a specific block from another characteristic. Although this sounds simple enough, identifying and satisfying the constraints when choosing values can be difficult.

How constraints are handled when values are selected depends on the coverage criterion chosen, and the decision is usually made when values are chosen. For the ACoC, PWC, and TWC criteria, the only reasonable option is to drop the infeasible pairs from consideration. For example, if PWC requires a particular pair that is not feasible, no amount of tinkering on the test engineer's part can make that requirement feasible. However, the situation is quite different for a criterion such as BCC. If a particular variation (for example, "less than zero" for "Relation of Side 1 to zero") conflicts with the base case (for example, "scalene" for "Geometric Classification"), then the obvious thing to do is change the offending choice for the base case so that the variation *is* feasible. In this case, "Geometric Classification" clearly needs to change to "invalid."

As another example, consider sorting an array. The input to our sort routine will be a variable length array of some arbitrary type. The output will have three parts: (1) a permutation of the input array, sorted in ascending order, (2) the largest value (max), and (3) the smallest value (min). We might consider the following characteristics:

- Length of array
- Type of elements

- Max value
- Min value
- Position of max value
- Position of min value

These characteristics can, in turn, reasonably result in the partitioning summarized as follows:

Length	{0, 1, 2..100, 101..MAXINT}
Type	{int, char, string, other}
Max	{≤ 0, 1, > 1, 'a', 'Z', 'b', . . . , 'Y', blank, nonblank}
Min	{ · · · }
Max pos	{1, 2..Length-1, Length}
Min pos	{1, 2..Length-1, Length}

The discerning reader will of course notice that not all combinations are possible. For example, if *Length* = 0, then nothing else matters. Also, some of the *Max* and *Min* values are available only if *Type* = *int*, and others if *Type* = *char*.

4.4 BIBLIOGRAPHIC NOTES

In the research literature, several testing methods have been described that are generally based on the model that the input space of the test object should be divided into subsets, with the assumption that all inputs in the same subset cause similar behavior. These are collectively called *partition testing* and include equivalence partitioning [249], boundary value analysis [249], category partition [283], and domain testing [29]. An extensive survey with examples was published by Grindal et al. [143].

The derivation of partitions and values started with Balcer, Hasling, and Ostrand's category partition method in 1988 [23, 283]. An alternate visualization is that of classification trees introduced by Grochtman, Grimm, and Wegener in 1993 [145, 146]. Classification trees organize the input space partitioning information into a tree structure. The first level nodes are the parameters and environment variables (characteristics); they may be recursively broken into sub-categories. Blocks appear as leaves in the tree and combinations are chosen by selecting among the leaves.

Chen et al. empirically identified common mistakes that testers made during input parameter-modeling [64]. Many of the concepts on input domain modeling in this chapter come from Grindal's PhD work [140, 142, 144]. Both Cohen et al. [84] and Yin et al. [363] suggest functionality oriented approaches to input parameter modeling. Functionality-oriented input parameter modeling was also implicitly used by Grindal et al. [141]. Two other IDM-related methods are Classification Trees [145] and a UML activity diagram based method [65]. Beizer [29], Malaiya [222], and Chen et al. [64] also address the problem of characteristic selection.

Grindal published an analytical/empirical comparison of different constraint-handling mechanisms [144].

Stocks and Carrington [320] provided a formal notion of specification-based testing that encompasses most approaches to input space partition testing. In particular,

they addressed the problem of refining test frames (which we simply and informally call test requirements in this book) to test cases.

The each choice and base choice criteria were introduced by Ammann and Offutt in 1994 [16]. Cohen et al. [84] indicated that valid and invalid parameter values should be treated differently with respect to coverage. *Valid* values lie within the bounds of normal operation of the test object, and *invalid* values lie outside the normal operating range. Invalid values often result in an error message and the execution terminates. To avoid one invalid value masking another, Cohen et al. suggested that only one invalid value should be included in each test case.

Burroughs et al. [58] and Cohen et al. [84, 85, 86] suggested the heuristic pair-wise coverage as part of the Automatic Efficient Test Generator (AETG). AETG also includes a variation on the base choice combination criterion. In AETG's version, called "default testing," the tester varies the values of one characteristic at a time while the other characteristics contain *some* default value. The term "default testing" was also used by Burr and Young [57], who described yet another variation of the base choices. In their version, all characteristics except one contain the default value, and the remaining characteristics contain a maximum or a minimum value. This variant will not necessarily satisfy "each choice coverage."

The Constrained Array Test System (CATS) tool for generating test cases was described by Sherwood [313] to satisfy pair-wise coverage. For programs with two or more characteristics, the in-parameter-order (IPO) combination strategy [205, 206, 322] generates a test suite that satisfies pair-wise coverage for the first two parameters (characteristic in our terminology). The test suite is then extended to satisfy pair-wise coverage for the first three parameters and continues for each additional parameter until all parameters are included.

Williams and Probert invented *T*-wise coverage [354]. A special case of T-wise coverage called *variable strength* was proposed by Cohen, Gibbons, Mugridge, and Colburn [87]. This strategy requires higher coverage among a subset of characteristics and lower coverage across the others. Assume for example a test problem with four parameters A, B, C, D. Variable strength may require 3-wise coverage for parameters B, C, D and 2-wise coverage for parameter A. Cohen, Gibbons, Mugridge, and Colburn [87] suggested using simulated annealing (SA) to generate test suites for T-wise coverage. Shiba, Tsuchiya, and Kikuno [314] proposed using a genetic algorithm (GA) to satisfy pair-wise coverage. The same paper also suggested using the ant colony algorithm (ACA).

Mandl suggested using orthogonal arrays to generate values for T-wise coverage [224]. This idea was further developed by Williams and Probert [353]. Covering arrays [352] is an extension of orthogonal arrays. A property of orthogonal arrays is that they are *balanced*, which means that each characteristic value occurs the same number of times in the test suite. If only T-wise (for instance pair-wise) coverage is desired, the balance property is unnecessary and will make the algorithm less efficient. In a covering array that satisfies T-wise coverage, each T-tuple occurs at least once but not necessarily the same number of times. Another problem with orthogonal arrays is that for some problem sizes we do not have enough orthogonal arrays to represent the entire problem. This problem is also avoided by using covering arrays.

Several papers have provided experiential and experimental results of using input space partitioning. Heller [156] uses a realistic example to show that testing all

combinations of characteristic values is infeasible in practice. Heller concludes that we need to identify a subset of combinations of manageable size.

Kuhn and Reilly [195] investigated 365 error reports from two large real-life projects and discovered that pair-wise coverage was nearly as effective at finding faults as testing all combinations. More supporting data were given by Kuhn and Wallace [196].

Piwowarski, Ohba, and Caruso [291] describe how to apply code coverage successfully as a stopping criterion during functional testing. The authors formulated functional testing as the problem of selecting test cases from all combinations of values of the input parameters. Burr and Young [57] show that continually monitoring code coverage helps improve the input domain model. Initial experiments showed that ad hoc testing resulted in about 50% decision coverage, but by continually applying code coverage and refining the input domain models, decision coverage was increased to 84%.

Plenty of examples of applying input space partitioning in practice have been published. Dalal, Jain, Karunanithi, Leaton, Lott, Patton, and Horowitz [91, 92] report results from using the AETG tool. It was used to generate test cases for Bellcore's Intelligent Service Control Point, a rule-based system used to assign work requests to technicians, and a GUI window in a large application. Previously, Cohen, Dalal, Kajla, and Patton [85] demonstrated the use of AETG for screen testing, by testing the input fields for consistency and validity across a number of screens.

Burr and Young [57] also used the AETG tool to test a Nortel application that converts email messages from one format to another. Huller [171] used an IPO related algorithm to test ground systems for satellite communications.

Williams and Probert [353] demonstrated how input space partitioning can be used to organize configuration testing. Yilmaz, Cohen, and Porter [362] used covering arrays as a starting point for fault localization in complex configuration spaces.

Huller [171] showed that pair-wise configuration testing can save more than 60% in both cost and time compared to quasi-exhaustive testing. Brownlie, Prowse, and Phadke [50] compared the results of using orthogonal arrays (OA) on one version of a PMX/StarMAIL release with the results from conventional testing on a prior release. The authors estimated that 22% more faults would have been found if OA had been used on the first version.

Several studies have compared the number of tests generated. The number of tests varies when using nondeterministic algorithms. Several papers compared input space partitioning strategies that satisfy 2-wise or 3-wise coverage: IPO and AETG [205], OA and AETG [141], covering arrays (CA) and IPO [352], and AETG, IPO, SA, GA, and ACA [87, 314]. Most of them found very little difference.

Another way to compare algorithms is with respect to the execution time. Lei and Tai [206] showed that the time complexity of IPO is superior to that of AETG. Williams [352] reported that CA outperforms IPO by almost three orders of magnitude for the largest test problems in his study.

Grindal et al. [141] compared algorithms by the number of faults found. They found that BCC performs as well as AETG and OA despite fewer test cases.

Input space partitioning strategies can also be compared based on their code coverage. Cohen et al. [86] found that test suites generated by AETG for 2-wise

coverage reach over 90% block coverage. Burr and Young [57] got similar results for AETG, getting 93% block coverage with 47 test cases, compared with 85% block coverage for a restricted version of BCC using 72 test cases.

NOTES

1 We choose to use blocks for simplicity. These are also sometimes called "partitions" in the literature.

5

Syntax-Based Testing

In previous chapters, we learned how to generate tests from graphs, logical expressions, and partitions of the input space. A fourth major source for test coverage criteria is syntactic descriptions of software artifacts. As with graphs and logical expressions, several types of artifacts can be used, including source and input requirements.

The essential characteristic of syntax-based testing is that a syntactic description such as a grammar or BNF is used. Chapters 2 and 3 discussed how to build graph models and logic models from artifacts such as the program, design descriptions, and specifications. Chapter 4 discussed how to build a model of the inputs based on some description of the input space. Then test criteria were applied to the models. With syntax-based testing, however, the syntax of the software artifact is used as the model and tests are created from the syntax.

5.1 SYNTAX-BASED COVERAGE CRITERIA

Syntax structures can be used for testing in several ways. We can use the syntax to generate artifacts that are valid (correct syntax), or artifacts that are invalid (incorrect syntax). Sometimes the structures we generate are test cases themselves, and sometimes they are used to help us find test cases. We explore these differences in the subsections of this chapter. As usual, we begin by defining general criteria on syntactic structures and then make them specific to specific artifacts.

5.1.1 BNF Coverage Criteria

It is very common in software engineering to use structures from automata theory to describe the syntax of software artifacts. Programming languages are described in BNF grammar notation, program behavior is described in finite state machines, and allowable inputs to programs are defined by grammars. Regular expressions and context free grammars are especially useful. Consider the regular expression:

$$(G\ s\ n \mid B\ t\ n)^*$$

The star is a "closure" operator that indicates zero or more occurrences of the expression it modifies. The vertical bar is the "choice" operator, and indicates either choice can be taken. Thus, this regular expression describes any sequence of "G s n" and "B t n." G and B may be commands to a program and s, t, and n may be arguments, method calls with parameters, or messages with values. The arguments s, t, and n can be literals, or represent a large set of values, for example, numbers or strings.

A test case can be a sequence of strings that satisfy the regular expression. For example, if the arguments are supposed to be numbers, the following may represent one test with four components, two separate tests, three separate tests, or four separate tests:

```
G 17 08.01.90
B 13 06.27.94
G 13 11.21.94
B 04 01.09.03
```

Although regular expressions are sometimes sufficient, a more expressive grammar is often used. The prior example can be refined into a grammar form as follows:

```
stream ::= action*
action ::= actG | actB
actG   ::= "G" s n
actB   ::= "B" t n
s      ::= digit^1-3
t      ::= digit^1-3
n      ::= digit^2 "." digit^2 "." digit^2
digit  ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

A grammar has a special symbol called the *start symbol*. In this case, the start symbol is stream. Symbols in the grammar are either *nonterminals*, which must be rewritten further, or *terminals*, for which no rewriting is possible. In the example, the symbols on the left of the ::= sign are all nonterminals, and everything in quotes is a terminal. Each possible rewriting of a given nonterminal is called a *production* or *rule*. In this grammar, a star superscript means zero or more, a plus superscript means one or more, a numeric superscript indicates the required number of repetitions, and a numeric range ($a - b$) means there has to be at least a repetitions, and no more than b.

Grammars can be used in two ways. A *recognizer*, as defined in Chapter 1, decides whether a given string (or test case) is in the grammar. This is the classical automata theory problem of parsing, and automated tools (such as the venerable lex and yacc) make the construction of recognizers very easy. Recognizers are extremely useful in testing, because they make it possible to decide if a given test case is in a particular grammar or not. The other use of grammars is to build *generators*, also defined in Chapter 1. A generator derives a string of terminals from the grammar

start symbol. In this case, the strings are test inputs. For example, the following derivation results in the test case above.

```
stream → action^*
       → action action^*
       → actG action^*
       → G s n action^*
       → G digit^(1-3) digit^2 . digit^2 . digit^2 action^*
       → G digitdigit digitdigit.digitdigit.digitdigit action^*
       → G 14 08.01.90 action^*
       ⋮
```

The derivation proceeds by systematically replacing the next nonterminal (in this case, "action^*") with one of its productions. Derivation continues until all nonterminals have been rewritten and only terminal symbols remain. The key to testing is which derivations should be used, and this is how criteria are defined on grammars.

Although many test criteria could be defined, the most common and straightforward are *terminal symbol coverage* and *production coverage*.

CRITERION **5.29 Terminal Symbol Coverage (TSC):** *TR contains each terminal symbol t in the grammar G.*

CRITERION **5.30 Production Coverage (PDC):** *TR contains each production p in the grammar G.*

By now, it should be easy to see that PDC subsumes TSC (if we cover every production, we cover every terminal symbol). Some readers may also note that grammars and graphs have a natural relationship. Therefore, TSC and PDC can be rewritten to be equivalent to node coverage and edge coverage on the graph that represents the grammar. Of course, this means that the other graph-based coverage criteria can also be defined on grammars. To our knowledge, neither researchers nor practitioners have taken this step.

The only other related criterion defined here is the impractical one of deriving all possible strings in a graph.

CRITERION **5.31 Derivation Coverage (DC):** *TR contains every possible string that can be derived from the grammar G.*

The number of tests generated by TSC will be bounded by the number of terminal symbols. The stream BNF above has 13 terminal symbols: G, B, ., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. It has 18 productions (note the '|' symbol adds productions, so "action" has two productions and "digit" has 10). The number of derivations for DC depends on the details of the grammar, but generally can be infinite. If we ignore the first production in the stream BNF, we have a finite number of derivable

strings. Two possible actions are (G and B), s and t each has a maximum of three digits with 10 choices, or 1000. The n nonterminal has three sets of two digits with 10 choices apiece, or 10^6. Altogether, the stream grammar can generate $2 * 1000 * 10^6 = 2,000,000,000$ strings. DC is of theoretical interest but is obviously impractical. (A point to remember the next time a tool salesperson or job applicant claims to have done "full string coverage" or "path coverage.")

TSC, PDC, and DC generate test cases that are members of the set of strings defined by the grammar. It is sometimes very helpful to generate test cases that are **not** in the grammar, which is addressed by the criteria in the next subsection.

EXERCISES

Section 5.1.1.

1. Consider how often the idea of covering nodes and edges pops up in software testing. Write a short essay to explain this.
2. Just as with graphs, it is possible to generate an infinite number of tests from a grammar. How and what makes this possible?

5.1.2 Mutation Testing

One of the interesting things that grammars do is describe what an input is *not*. We say that an input is *valid* if it is in the language specified by the grammar, and *invalid* otherwise. For example, it is quite common to require a program to reject malformed inputs, and this property should clearly be tested, since it is easy for programmers to forget it or get it wrong.

Thus, it is often useful to produce invalid strings from a grammar. It is also helpful in testing to use strings that are *valid* but that follow a different derivation from a preexisting string. Both of these strings are called *mutants*.[1] This can be done by mutating the grammar, then generating strings, or by mutating values during a production.

Mutation can be applied to various artifacts, as discussed in the following subsections. However, it has primarily been used as a program-based testing method, and much of the theory and many of the detailed concepts are specific to program-based mutation. Therefore, a lot more details appear in Section 5.2.2.

Mutation is always based on a set of "mutation operators," which are expressed with respect to a "ground" string.

Definition 5.45 Ground string: A string that is in the grammar.

Definition 5.46 Mutation Operator: A rule that specifies syntactic variations of strings generated from a grammar.

Definition 5.47 Mutant: The result of one application of a mutation operator.

Mutation operators are usually applied to ground strings, but can also be applied to a grammar, or dynamically during a derivation. The notion of a mutation operator is extremely general, and so a very important part of applying mutation to any artifact is the design of suitable mutation operators. A well designed set of

operators can result in very powerful testing, but a poorly designed set can result in ineffective tests. For example, a commercial tool that "implements mutation" but that only changes predicates to *true* and *false* would simply be an expensive way to implement branch coverage.

We sometimes have a particular ground string in mind, and sometimes the ground string is simply the implicit result of not applying any mutation operators. For example, we care about the ground string when applying mutation to program statements. The ground string is the sequence of program statements in the program under test, and the mutants are slight syntactic variations of that program. We do not care about the ground string during invalid input testing, when the goal is to see if a program correctly responds to invalid inputs. The ground strings are valid inputs, and variants are the invalid inputs. For example, a valid input might be a transaction request from a correctly logged-in user. The invalid version might be the same transaction request from a user who is not logged in.

Consider the grammar in Section 5.1.1. If the first string shown, `G 17 08.01.90`, is taken as a ground string, two *valid* mutants may be

```
B 17 08.01.90
G 43 08.01.90
```

Two *invalid* mutants may be

```
12 17 08.01.90
G 23 08.01
```

When the ground string does not matter, mutants can be created directly from the grammar by modifying productions during a derivation, using a generator approach as introduced in the previous section. That is, if the ground strings are not of direct interest, they do not need to be explicitly generated.

When applying mutation operators, two issues often come up. First, should more than one mutation operator be applied at the same time to create one mutant? That is, should a mutated string contain one mutated element, or several? Common sense indicates no, and strong experimental and theoretical evidence has been found for mutating only one element at a time in program-based mutation. Another question is should every possible application of a mutation operator to a ground string be considered? This is usually done in program-based mutation. One theoretical reason is that program-based mutation subsumes a number of other test criteria, and if operators are not applied comprehensively, then that subsumption is lost. However, this is not always done when the ground string does not matter, for example, in the case of invalid input testing. This question is explored in more detail in the following application subsections.

Mutation operators have been designed for several programming languages, formal specification languages, BNF grammars, and at least one data definition language (XML). For a given artifact, the set of mutants is M and each mutant $m \in M$ will lead to a test requirement.

When a derivation is mutated to produce valid strings, the testing goal is to "kill" the mutants by causing the mutant to produce different output. More formally, given a mutant $m \in M$ for a derivation D and a test t, t is said to *kill* m if and only if the output of t on D is different from the output of t on m. The derivation D may be represented by the complete list of productions followed, or it may simply be represented by the final string. For example, in Section 5.2.2, the strings are programs or program components. Coverage is defined in terms of killing mutants.

CRITERION **5.32 Mutation Coverage (MC):** *For each mutant $m \in M$, TR contains exactly one requirement, to* kill *m.*

Thus, coverage in mutation equates to killing the mutants. The amount of coverage is usually written as a percent of mutants killed and called the *mutation score*.

When a grammar is mutated to produce invalid strings, the testing goal is to run the mutants to see if the behavior is correct. The coverage criterion is therefore simpler, as the mutation operators are the test requirements.

CRITERION **5.33 Mutation Operator Coverage (MOC):** *For each mutation operator, TR contains exactly one requirement, to create a mutated string m that is derived using the mutation operator.*

CRITERION **5.34 Mutation Production Coverage (MPC):** *For each mutation operator, and each production that the operator can be applied to, TR contains the requirement to create a mutated string from that production.*

The number of test requirements for mutation is somewhat difficult to quantify because it depends on the syntax of the artifact as well as the mutation operators. In most situations, mutation yields more test requirements than any other test criterion. Subsequent sections have more data on quantifying specific collections of mutation operators and more details are in the bibliographic notes.

Mutation testing is also difficult to apply by hand, and automation is more complicated than for most other criteria. As a result, mutation is widely considered a "high-end" test criterion, more effective than most but also more expensive. One common use of mutation is as a sort of "gold standard" in experimental studies for comparative evaluation of other test criteria.

EXERCISES

Section 5.1.2.

1. Define mutation score.
2. How is the mutation score related to coverage from Chapter 1?
3. Consider the stream BNF in Section 5.1.1 and the ground string "`B 10 06.27.94`." Give three valid and three invalid mutants of the string.

The rest of this chapter explores various forms of BNF and mutation testing. The table below summarizes the sections and the characteristics of the various flavors of syntax testing. Whether the use of syntax testing creates valid or invalid tests is noted

for both BNF and mutation testing. For mutation testing, we also note whether a ground string is used, whether the mutants are tests or not, and whether mutants are killed.

	Program-based	Integration	Specification-based	Input space
BNF	5.2.1	5.3.1	5.4.1	5.5.1
Grammar	Programming languages	No known applications	Algebraic specifications	Input languages, including XML
Summary	Compiler testing			Input space testing
Mutation	5.2.2	5.3.2	5.4.2	5.5.2
Grammar	Programming languages	Programming languages	FSMs	Input languages, including XML
Summary	Mutates programs	Tests integration	Uses model-checking	Error checking
Ground?	Yes	Yes	Yes	No
Valid?	Yes, must compile	Yes, must compile	Yes	No
Tests?	Mutants are not tests	Mutants are not tests	Mutants are not tests	Mutants are tests
Killing?	Yes	Yes	Yes	No notion of killing
Notes	Strong and weak mutants. Subsumes many other techniques.	Includes object-oriented testing	Automatic detection of equivalent mutants	Sometimes the grammar is mutated, then strings are produced

5.2 PROGRAM-BASED GRAMMARS

As with most criteria, syntax-based testing criteria have been applied to programs more than other artifacts. The BNF coverage criteria have been used to generate programs to test compilers. Mutation testing has been applied to methods (unit testing) and to classes (integration testing). Application to classes is discussed in the next section.

5.2.1 BNF Grammars for Languages

The primary purpose of BNF testing for languages has been to generate test suites for compilers. As this is a very specialized application, we choose not to dwell on it in this book. The bibliographic notes section has pointers to the relevant, and rather old, literature.

5.2.2 Program-Based Mutation

Mutation was originally developed for programs and this section has significantly more depth than other sections in this chapter. Program-based mutation uses operators that are defined in terms of the grammar of a particular programming language. We start with a **ground string**, which is a program that is being tested. We then apply mutation operators to create mutants. These mutants must be compilable, so program-based mutation creates **valid** strings. The mutants are not tests, but are used to help us find tests.

Given a ground string program or method, a mutation-adequate test set distinguishes the program from a set of syntactic variations, or mutants, of that program. A simple example of a mutation operator for a program is the *Arithmetic Operation*

Original Method	With Embedded Mutants

int Min (int A, int B)
{
  int minVal;
  minVal = A;
  if (B < A)
  {
  minVal = B;
  }
  return (minVal);
} // end Min
	int Min (int A, int B)
	{
	  int minVal;
	  minVal = A;
	Δ1 minVal = **B**;
	  if (B < A)
	Δ2 if (B > A)
	Δ3 if (B < **minVal**)
	  {
	  minVal = B;
	Δ4 **Bomb()**;
	Δ5 minVal = **A**;
	Δ6 minVal = **failOnZero (B)**;
	  }
	  return (minVal);
	} // end Min

Figure 5.1. Method Min and six mutants.

Mutation operator, which changes an assignment statement like "`x = a + b`" into a variety of possible alternatives, including "`x = a - b`", "`x = a * b`", and "`x = a / b`". Unless the assignment statement appears in a very strange program, it probably matters which arithmetic operator is used, and a decent test set should be able to distinguish among the various possibilities. It turns out that by careful selection of the mutation operators, a tester can develop very powerful test sets.

Mutation testing is used to help the user to strengthen the quality of test data iteratively. Test data are used to evaluate the ground program with the goal of causing each mutant to exhibit different behavior. When this happens, the mutant is considered *dead* and no longer needs to remain in the testing process since the fault that it represents will be detected by the same test that killed it. More importantly, the mutant has satisfied its requirement of identifying a useful test case.

A key to successful use of mutation is the mutation operators, which are designed for each programming, specification, or design language. In program-based mutation, invalid strings are syntactically illegal and would be caught by a compiler. These are called *stillborn* mutants and should not be generated. A *trivial* mutant can be killed by almost any test case. Some mutants are functionally *equivalent* to the original program. That is, they always produce the same output as the original program, so no test case can kill them. Equivalent mutants represent infeasible test requirements, as discussed in the previous chapters.

We refine the notion of killing and coverage for program-based mutation. These definitions are consistent with the previous section.

> *Definition 5.48 Killing Mutants:* Given a mutant $m \in M$ for a ground string program P and a test t, t is said to *kill* m if and only if the output of t on P is different from the output of t on m.

As said in Section 5.1.2, it is hard to quantify the number of test requirements for mutation. In fact, it depends on the specific set of operators used and the language that the operators are applied to. One of the most widely used mutation systems was Mothra. It generated 44 mutants for the Fortran version of the `Min()` method in Figure 5.1. For most collections of operators, the number of program-based mutants is roughly proportional to the product of the number of references to variables times

the number of variables that are declared ($O(Refs*Vars)$). The selective mutation approach mentioned below under "Designing Mutation Operators" eliminates the number of data objects so that the number of mutants is proportional to the number of variable references ($O(Refs)$). More details are in the bibliographic notes.

Program-based mutation has traditionally been applied to individual statements for unit level testing. Figure 5.1 contains a small Java method with six mutated lines (each preceded by the Δ symbol). Note that each mutated statement represents a separate program. The mutation operators are defined to satisfy one of two goals. One goal is to mimic typical programmer mistakes, thus trying to ensure that the tests can detect those mistakes. The other goal is to force the tester to create tests that have been found to effectively test software. In Figure 5.1, mutants 1, 3, and 5 replace one variable reference with another, mutant 2 changes a relational operator, and mutant 4 is a special mutation operator that causes a runtime failure as soon as the statement is reached. This forces every statement to be executed, thus getting statement or node coverage.

Mutant 6 looks unusual, as the operator is intended to force the tester to create an effective test. The *failOnZero*() method is a special mutation operator that causes a failure if the parameter is zero and does nothing if the parameter is not zero (it returns the value of the parameter). Thus, mutant 6 can be killed only if **B** has the value zero, which forces the tester to follow the time-tested heuristic of causing every variable and expression to have the value of zero.

One point that is sometimes confusing about mutation is how tests are created. When applying program-based mutation, the direct goal of the tester is to kill mutants; an indirect goal is to create good tests. Even less directly, the tester wants to find faults. Tests that kill mutants can be found by intuition, or if more rigor is needed, by analyzing the conditions under which a mutant will be killed.

The RIP fault/failure model was discussed in Section 1.2. Program-based mutations represent a software failure by a mutant, and reachability, infection, and propagation refer to reaching the mutant, the mutant causing the program state to be incorrect, and the eventual output of the program to be incorrect.

Weak mutation relaxes the definition of "killing" a mutant to include only reachability and infection, but **not** propagation. That is, weak mutation checks the internal state of the program immediately after execution of the mutated component (that is, after the expression, statement, or basic block). If the state is incorrect the mutant is killed. This is weaker than standard (or *strong*) mutation because an incorrect state does **not** always propagate to the output. That is, strong mutation may require more tests to satisfy coverage than weak mutation. Experimentation has shown that the difference is very small in most cases.

This difference can be formalized by refining the definition of killing mutants given previously.

Definition 5.49 Strongly Killing Mutants: Given a mutant $m \in M$ for a program P and a test t, t is said to *strongly kill* m if and only if the output of t on P is different from the output of t on m.

CRITERION **5.35 Strong Mutation Coverage (SMC):** *For each $m \in M$, TR contains exactly one requirement, to strongly kill m.*

Definition 5.50 Weakly Killing Mutants: Given a mutant $m \in M$ that modifies a location l in a program P, and a test t, t is said to *weakly kill* m if and only if the state of the execution of P on t is different from the state of the execution of m immediately after l.

CRITERION **5.36 Weak Mutation Coverage (WMC):** *For each $m \in M$, TR contains exactly one requirement, to weakly kill m.*

Consider mutant 1 in Figure 5.1. The mutant is on the first statement, thus the reachability condition is always satisfied (*true*). In order to infect, the value of **B** must be different from the value of **A**, which is formalized as $(A \neq B)$. To propagate, the mutated version of **Min** must return an incorrect value. In this case, **Min** must return the value that was assigned in the first statement, which means that the statement inside the **if** block must **not** be executed. That is, $(B < A) = false$. The complete test specification to kill mutant 1 is

$$\begin{aligned}
&\text{Reachability} : true \\
&\text{Infection} : A \neq B \\
&\text{Propagation} : (B < A) = false \\
&\text{Full Test Specification} : true \wedge (A \neq B) \wedge ((B < A) = false) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \equiv (A \neq B) \wedge (B \geq A) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \equiv (B > A)
\end{aligned}$$

Thus, the test case value $(A = 5, B = 7)$ should cause mutant 1 to result in a failure. The original method will return the value 5 (A) but the mutated version returns 7.

Mutant 3 is an example of an equivalent mutant. Intuitively, **minVal** and **A** have the same value at that point in the program, so replacing one with the other has no effect. As with mutant 1, the reachability condition is *true*. The infection condition is $(B < A) \neq (B < minVal)$. However, dutiful analysis can reveal the assertion $(minVal = A)$, leading to the combined condition $((B < A) \neq (B < minVal)) \wedge (minVal = A)$. Simplifying by eliminating the inequality $\neq$ gives

$$(((B < A) \wedge (B \geq minVal)) \vee ((B \geq A) \wedge (B < minVal))) \wedge (minVal = A)$$

Rearranging the terms gives

$$(((A > B) \wedge (B \geq minVal)) \vee ((A \leq B) \wedge (B < minVal))) \wedge (minVal = A)$$

If $(A > B)$ and $(B \geq minVal)$, then by transitivity, $(A > minVal)$. Applying transitivity to both the first two disjuncts gives

$$((A > minVal) \vee (A < minVal)) \wedge (minVal = A)$$

Finally, the first disjunct can be reduced to a simple inequality, resulting in the following contradiction:

$$(A \neq minVal) \wedge (minVal = A)$$

The contradiction means that no values exist that can satisfy the conditions, thus the mutant is provably equivalent. In general, detecting equivalent mutants, just like detecting infeasible paths, is an undecidable problem. However, strategies

such as algebraic manipulations and program slicing can detect some equivalent mutants.

As a final example, consider the following method, with one mutant shown embedded in statement 4:

```
1      boolean isEven (int X)
2      {
3        if (X < 0)
4          X = 0 - X;
Δ4         X = 0;
5        if (float) (X/2) == ((float) X) / 2.0
6          return (true);
7        else
8          return (false);
9      }
```

The reachability condition for mutant $\Delta 4$ is $(X < 0)$ and the infection condition is $(X \neq 0)$. If the test case `X = -6` is given, then the value of `X` after statement 4 is executed is 6 and the value of `X` after the **mutated** version of statement 4 is executed is 0. Thus, this test satisfies reachability and infection, and the mutant will be killed under the weak mutation criterion. However, 6 and 0 are both even, so the decision starting on statement 5 will return true for both the mutated and nonmutated versions. That is, propagation is not satisfied, so test case `X = -6` will not kill the mutant under the strong mutation criterion. The propagation condition for this mutant is that the number be odd. Thus, to satisfy the strong mutation criterion, we require $(X < 0) \wedge (X \neq 0) \wedge odd(X)$, which can be simplified to X is an odd, negative integer.

Testing Programs with Mutation

A test process gives a sequence of steps to follow to generate test cases. A single criterion may be used with many processes, and a test process may not even include a criterion. Choosing a test process for mutation is particularly difficult because mutation analysis is actually a way to measure the quality of the test cases and the actual testing of the software is a side effect. In practical terms, however, the software is tested, and tested well, or the test cases do not kill mutants. This point can best be understood by examining a typical mutation analysis process.

Figure 5.2 shows how mutation testing can be applied. The tester submits the program under test to an automated system, which starts by creating mutants. Optionally, those mutants are then analyzed by a heuristic that detects and eliminates as many equivalent mutants as possible.[2] A set of test cases is then generated automatically and executed first against the original program, and then the mutants. If the output of a mutant program differs from the original (correct) output, the mutant is marked as being dead and is considered to have been *strongly killed* by that test case. Dead mutants are not executed against subsequent test cases. Test cases that do not strongly kill at least one mutant are considered to be "ineffective" and eliminated, even though such test cases may weakly kill one or more mutants. This

Figure 5.2. Mutation testing process.
Bold boxes represent steps that are automated; other boxes represent manual steps.

is because the requirement stated above requires the output (and not the internal state) to be different.

Once all test cases have been executed, coverage is computed as a mutation score. The mutation score is the ratio of dead mutants over the total number of non-equivalent mutants. If the mutation score reaches 1.00, that means all mutants have been detected. A test set that kills all the mutants is said to be *adequate* relative to the mutants.

A mutation score of 1.00 is usually impractical, so the tester defines a "threshold" value, which is a minimum acceptable mutation score. If the threshold has not been reached, then the process is repeated, each time generating test cases to target live mutants, until the threshold mutation score is reached. Up to this point, the process has been entirely automatic. To finish testing, the tester will examine expected output of the effective test cases, and fix the program if any faults are found. This leads to the fundamental premise of mutation testing: **In practice, if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault**.

Designing Mutation Operators

Mutation operators must be chosen for each language and although they overlap quite a bit, some differences are particular to the language, often depending on the language features. Researchers have designed mutation operators for various programming languages, including Fortran IV, COBOL, Fortran 77, C, C integration testing, Lisp, Ada, Java, and Java class relationships. Researchers have also designed Mutation operators for the formal specification language SMV (discussed in Section 5.4.2), and for XML messages (discussed in Section 5.5.2).

As a field, we have learned a lot about designing mutation operators over the years. Detailed lists of mutation operators for various languages are provided in

the literature, as referenced in the bibliographic notes for this chapter. Mutation operators are generally designed either to mimic typical programmer mistakes, or to encourage testers to follow common testing heuristics. Operators that change relational operators or variable references are examples of operators that mimic typical programmer mistakes. The *failOnZero*() operator used in Figure 5.1 is an example of the latter design; the tester is encouraged to follow the common testing heuristic of "causing each expression to become zero."

When first designing mutation operators for a new language, it is reasonable to be "inclusive," that is, include as many operators as possible. However, this often results in a large number of mutation operators, and an even larger number of mutants. Researchers have devoted a lot of effort to trying to find ways to use fewer mutants and mutation operators. The two most common ways to have fewer mutants are (1) to randomly sample from the total number of mutants, and (2) to use mutation operators that are particularly effective.

The term *selective mutation* has been used to describe the strategy of using only mutation operators that are particularly effective. Effectiveness has been evaluated as follows: if tests that are created specifically to kill mutants created by mutation operator o_i also kill mutants created by mutation operator o_j with very high probability, then mutation operator o_i is more *effective* than o_j.

This notion can be extended to consider a collection of effective mutation operators as follows:

> *Definition 5.51 Effective Mutation Operators:* If tests that are created specifically to kill mutants created by a collection of mutation operators $O = \{o_1, o_2, \ldots\}$ also kill mutants created by all remaining mutation operators with very high probability, then O defines an effective set of mutation operators.

Researchers have concluded that a collection of mutation operators that insert unary operators and that modify unary and binary operators will be **effective**. The actual research was done with Fortran 77 (the Mothra system), but the results are adapted to Java in this chapter. Corresponding operators can be defined for other languages. The operators defined below are used throughout the remainder of this chapter as the defining set of program-level mutation operators.

1. ***ABS – Absolute Value Insertion***:

Each arithmetic expression (and subexpression) is modified by the functions *abs*(), *negAbs*(), and *failOnZero*().

abs() returns the absolute value of the expression and *negAbs*() returns the negative of the absolute value. *failOnZero*() tests whether the value of the expression is zero. If it is, the mutant is killed; otherwise, execution continues and the value of the expression is returned. This operator is designed specifically to force the tester to cause each numeric expression to have the value 0, a negative value, and a positive value. For example, the statement "`x = 3 * a;`" is mutated to create the following three statements:

```
x = 3 * abs (a);
x = 3 * - abs (a);
x = 3 * failOnZero (a);
```

2. ***AOR – Arithmetic Operator Replacement***:

Each occurrence of one of the arithmetic operators $+$, $-$, $*$, $/$, $**$, and % is replaced by each of the other operators. In addition, each is replaced by the special mutation operators *leftOp*, *rightOp*, and *mod*.

leftOp returns the left operand (the right is ignored), *rightOp* returns the right operand, and *mod* computes the remainder when the left operand is divided by the right. For example, the statement "`x = a + b;`" is mutated to create the following seven statements:

```
x = a - b;
x = a * b;
x = a / b;
x = a ** b;
x = a;
x = b;
x = a % b;
```

3. ***ROR – Relational Operator Replacement***:

Each occurrence of one of the relational operators ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$) is replaced by each of the other operators and by *falseOp* and *trueOp*.

falseOp always returns *false* and *trueOp* always returns *true*. For example, the statement "if (`m > n`)" is mutated to create the following seven statements:

```
if (m >= n)
if (m < n)
if (m <= n)
if (m == n)
if (m != n)
if (false)
if (true)
```

4. ***COR – Conditional Operator Replacement***:

Each occurrence of each logical operator (*and*–&&, *or*–||, *and with no conditional evaluation*–&, *or with no conditional evaluation*–|, *not equivalent* – ^) is replaced by each of the other operators; in addition, each is replaced by *falseOp*, *trueOp*, *leftOp*, and *rightOp*.

leftOp returns the left operand (the right is ignored) and *rightOp* returns the right operand. *falseOp* always returns *false* and *trueOp* always returns *true*. For example, the statement "if (`a && b`)" is mutated to create the following eight statements:

```
if (a || b)
if (a & b)
if (a | b)
if (a ^ b)
if (false)
if (true)
```

```
if (a)
if (b)
```

5. ***SOR – Shift Operator Replacement***:

Each occurrence of one of the shift operators <<, >>, and >>> is replaced by each of the other operators. In addition, each is replaced by the special mutation operator *leftOp*.

leftOp returns the left operand unshifted. For example, the statement "`x = m << a;`" is mutated to create the following three statements:

```
x = m >> a;
x = m >>> a;
x = m;
```

6. ***LOR – Logical Operator Replacement***:

Each occurrence of each bitwise logical operator (*bitwise and* (&), *bitwise or* (|), and *exclusive or* (^)) is replaced by each of the other operators; in addition, each is replaced by *leftOp* and *rightOp*.

leftOp returns the left operand (the right is ignored) and *rightOp* returns the right operand. For example, the statement "`x = m & n;`" is mutated to create the following four statements:

```
x = m | n;
x = m ^ n;
x = m;
x = n;
```

7. ***ASR – Assignment Operator Replacement***:

Each occurrence of one of the assignment operators (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=) is replaced by each of the other operators.

For example, the statement "`x += 3;`" is mutated to create the following ten statements:

```
x -= 3;
x *= 3;
x /= 3;
x %= 3;
x &= 3;
x |= 3;
x ^= 3;
x <<= 3;
x >>= 3;
x >>>= 3;
```

8. ***UOI – Unary Operator Insertion***:

Each unary operator (arithmetic +, arithmetic −, conditional !, logical ~) is inserted before each expression of the correct type.

For example, the statement "`x = 3 * a;`" is mutated to create the following four statements:

```
x = 3 * +a;
x = 3 * -a;
x = +3 * a;
x = -3 * a;
```

9. ***UOD – Unary Operator Deletion***:

Each unary operator (arithmetic +, arithmetic −, conditional !, logical ~) is deleted.

For example, the statement "`if !(a > -b)`" is mutated to create the following two statements:

```
if (a > -b)
if !(a > b)
```

Two other operators that are useful in examples are scalar variable replacement and the "bomb" operator. Scalar variable replacement results in a lot of mutants (V^2 if V is the number of variables), and it turns out that it is not necessary given the above operators. It is included here as a convenience for examples. The bomb operator results in only one mutant per statement, but it is also not necessary given the above operators.

10. ***SVR – Scalar Variable Replacement***:

Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.

For example, the statement "`x = a * b;`" is mutated to create the following six statements:

```
x = a * a;
a = a * b;
x = x * b;
x = a * x;
x = b * b;
b = a * b;
```

11. ***BSR—Bomb Statement Replacement***:

Each statement is replaced by a special *Bomb*() function.

Bomb() signals a failure as soon as it is executed, thus requiring the tester to reach each statement. For example, the statement "`x = a * b;`" is mutated to create the following statement:

```
Bomb();
```

Subsumption of Other Test Criteria (*Advanced Topic*)

Mutation is widely considered the strongest test criterion in terms of finding the most faults. It is also the most expensive. This section shows that mutation subsumes a number of other coverage criteria. The proofs are developed by showing that specific mutation operators impose requirements that are identical to a specific coverage criterion. For each specific requirement defined by a criterion, a single mutant is created that can be killed only by test cases that satisfy the requirement. Therefore, the coverage criterion is satisfied if and only if the mutants associated with the requirements for the criterion are killed. In this case, the mutation operators that ensure coverage of a criterion are said to *yield* the criterion. If a criterion is yielded by one or more mutation operators, then mutation testing subsumes the criterion. Although mutation operators vary by language and mutation analysis tool, this section uses common operators that are used in most implementations. It is also possible to design mutation operators to force mutation to subsume other testing criteria. Further details are given in the bibliographic notes.

This type of proof has one subtle problem. The condition coverage criteria impose only a **local** requirement; for example, edge coverage requires that each branch in the program be executed. Mutation, on the other hand, imposes **global** requirements in addition to local requirements. That is, mutation also requires that the mutant program produce incorrect output. For edge coverage, some specific mutants can be killed only if each branch is executed **and** the final output of the mutant is incorrect. On the one hand, this means that mutation imposes stronger requirements than the condition coverage criteria. On the other hand, and somewhat perversely, this also means that sometimes a test set that satisfies a coverage criteria will not kill all the associated mutants. Thus, mutation as defined earlier will not strictly subsume the condition coverage criteria.

This problem is solved by basing the subsumptions on *weak mutation*. In terms of subsuming other coverage criteria, weak mutation only imposes the local requirements. In weak mutation, mutants that are **not** equivalent at the infection stage but **are** equivalent at the propagation stage (that is, an incorrect state is masked or repaired) are left in the set of test cases, so that edge coverage is subsumed. It is precisely the fact that such test cases are removed that strong mutation does not subsume edge coverage.

Thus, this section shows that the coverage criteria are subsumed by weak mutation, not strong mutation.

Subsumption is shown for graph coverage criteria from Chapter 2 and logic coverage criteria from Chapter 3. Some mutation operators only make sense for program source statements whereas others can apply to arbitrary structures such as logical expressions. For example, one common mutation operator is to replace statements with "bombs" that immediately cause the program to terminate execution or raise an exception. This mutation can only be defined for program statements. Another common mutation operator is to replace relational operators ($<$, $>$, etc.) with other relational operators (the ROR operator). This kind of relational operator replacement can be applied to any logical expression, including guards FSMs.

Node coverage requires each statement or basic block in the program to be executed. The mutation operator that replaces statements with "bombs" yields node coverage. To kill these mutants, we are required to find test cases that reach each

		(T T)	(T F)	(F T)	(F F)
	$a \wedge b$	T	F	F	F
1	$true \wedge b$	T	F	$\boxed{\text{T}}$	F
2	$false \wedge b$	$\boxed{\text{F}}$	F	F	F
3	$a \wedge true$	T	$\boxed{\text{T}}$	F	F
4	$a \wedge false$	$\boxed{\text{F}}$	F	F	F

Figure 5.3. Partial truth table for ($a \wedge b$).

basic block. Since this is exactly the requirement of node coverage, this operator yields node coverage and mutation subsumes node coverage.

Edge coverage requires each edge in the control flow graph to be executed. A common mutation operator is to replace each predicate with both *true* and *false* (the ROR operator). To kill the *true* mutant, a test case must take the *false* branch, and to kill the *false* mutant, a test case must take the *true* branch. This operator forces each branch in the program to be executed, and thus it yields edge coverage and mutation subsumes edge coverage.

Clause coverage requires each clause to become both *true* and *false*. The ROR, COR, and LOR mutation operators will together replace each clause in each predicate with both *true* and *false*. To kill the *true* mutant, a test case must cause the clause (and also the full predicate) to be *false*, and to kill the *false* mutant, a test case must cause the clause (and also the full predicate) to be *true*. This is exactly the requirement for clause coverage. A simple way to illustrate this is with a modified form of a truth table.

Consider a predicate that has two clauses connected by an `AND`. Assume the predicate is ($a \wedge b$), where a and b are arbitrary boolean-valued clauses. The partial truth table in Figure 5.3 shows ($a \wedge b$) on the top line with the resulting value for each of the four combinations of values for a and b. Below the line are four mutations that replace each of a and b with *true* and *false*. To kill the mutants, the tester must choose an input (one of the four truth assignments on top of the table) that causes a result that is different from that of the original predicate. Consider mutant 1, $true \wedge b$. Mutant 1 has the same result as the original clause for three of the four truth assignments. Thus, to kill that mutant, the tester must use a test case input value that causes the truth assignment (F T), as shown in the box. Likewise, mutant 3, $a \wedge true$, can be killed only if the truth assignment (T F) is used. Thus, mutants 1 and 3 are killed if and only if clause coverage is satisfied, and the mutation operator yields clause coverage for this case. Note that mutants 2 and 4 are not needed to subsume clause coverage.

Although the proof technique of showing that mutation operators yield clause coverage on a case-by-case basis with the logical operators is straightforward and relatively easy to grasp, it is clumsy. More generally, assume a predicate p and a clause a, and the clause coverage requirements to test $p(a)$, which says that a must evaluate to both *true* and *false*. Consider the mutation $\Delta p(a \rightarrow true)$ (that is, the predicate where a is replaced by *true*). The only way to satisfy the infection condition for this mutant (and thus kill it) is to find a test case that causes a to take on the value of *false*. Likewise, the mutation $\Delta p(a \rightarrow false)$ can be killed only by a test case that

causes a to take on the value of *true*. Thus, in the general case, the mutation operator that replaces clauses with *true* and *false* yield clause coverage and is subsumed by mutation.

Combinatorial coverage requires that the clauses in a predicate evaluate to each possible combination of truth values. In the general case combinatorial coverage has 2^N requirements for a predicate with N clauses. Since no single or combination of mutation operators produces 2^N mutants, it is easy to see that mutation cannot subsume COC.

Active clause coverage requires that each clause c in a predicate p evaluates to *true* and *false* **and** determines the value of p. The first version in Chapter 3, general active clause coverage allows the values for other clauses in p to have different values when c is true and c is false. It is simple to show that mutation subsumes general active clause coverage; in fact, we already have.

To kill the mutant $\Delta p(a \rightarrow true)$, we must satisfy the infection condition by **causing** $p(a \rightarrow true)$ **to have a different value from** $p(a)$, that is, a must determine p. Likewise, to kill $\Delta p(a \rightarrow false)$, $p(a \rightarrow false)$ must have a different result from $p(a)$, that is, a must determine p. Since this is exactly the requirement of GACC, this operator yields node coverage and mutation subsumes general active clause coverage. Note that this is only true if the incorrect value in the mutated program propagates to the end of the expression, which is one interpretation of weak mutation.

Neither correlated active clause coverage nor restricted active clause coverage are subsumed by mutation operators. The reason is that both CACC and RACC require pairs of tests to have certain properties. In the case of CACC, the property is that the predicate outcome be different on the two tests associated with a particular clause. In the case of RACC, the property is that the minor clauses have exactly the same values on the two tests associated with a particular clause. Since each mutant is killed (or not) by a single test case, (as opposed to a pair of test cases), mutation analysis, at least as traditionally defined, cannot subsume criteria that impose relationships between pairs of test cases.

Researchers have not determined whether mutation subsumes the inactive clause coverage criteria.

All-defs data flow coverage requires that each definition of a variable reach at least one use. That is, for each definition of a variable X on node n, there must be a definition-clear subpath for X from n to a node or an edge with a use of X. The argument for subsumption is a little complicated for All-defs, and unlike the other arguments, All-defs requires that strong mutation be used.

A common mutation operator is to delete statements with the goal of forcing each statement in the program to make an impact on the output.[3] To show subsumption of All-defs, we restrict our attention to statements that contain variable definitions. Assume that the statement s_i contains a definition of a variable x, and m_i is the mutant that deletes s_i ($\Delta s_i \rightarrow null$). To kill m_i under strong mutation, a test case t must (1) cause the mutated statement to be reached (reachability), (2) cause the execution state of the program after execution of s_i to be incorrect (infection), and (3) cause the final output of the program to be incorrect (propagation). Any test case that reaches s_i will cause an incorrect execution state, because the mutated version of s_i will not assign a value to x. For the final output of the mutant to be incorrect, two cases occur. First, if x is an output variable, t must have caused

an execution of a subpath from the deleted definition of x to the output without an intervening definition (def-clear). Since the output is considered a use, this satisfies the criterion. Second, if x is not an output variable, then not defining x at s_i must result in an incorrect output state. This is possible only if x is used at some later point during execution without being redefined. Thus, t satisfies the all-defs criterion for the definition of x at s_i, and the mutation operator yields all-defs, ensuring that mutation subsumes all-defs.

It is possible to design a mutation operator specifically to subsume all-uses, but such an operator has never been published or used in any tool.

EXERCISES

Section 5.2.

1. Provide reachability conditions, infection conditions, propagation conditions, and test case values to kill mutants 2, 4, 5, and 6 in Figure 5.1.
2. Answer questions (a) through (d) for the mutant in the two methods, findVal() and sum().
   (a) If possible, find a test input that does **not** reach the mutant.
   (b) If possible, find a test input that satisfies reachability but **not infection** for the mutant.
   (c) If possible, find a test input that satisfies infection, but **not propagation** for the mutant.
   (d) If possible, find a test input that kills mutant m.

```
//Effects: If numbers null throw NullPointerException
//   else return LAST occurrence of val in numbers[]
//   If val not in numbers[] return -1
1. public static int findVal(int numbers[], int val)
2. {
3.    int findVal = -1;
4.
5.    for (int i=0; i<numbers.length; i++)
5'.// for (int i=(0+1); i<numbers.length; i++)
6.       if (numbers [i] == val)
7.          findVal = i;
8.    return (findVal);
9. }
```

```
//Effects: If x null throw NullPointerException
//   else return the sum of the values in x

1. public static int sum(int[] x)
2. {
3.    int s = 0;
4.    for (int i=0; i < x.length; i++) }
5.    {
6.       s = s + x[i];
6'.   // s = s - x[i]; //AOR
7.    }
8.    return s;
9.  }
```

3. Refer to the TestPat program in Chapter 2. Consider Mutant A and Mutant B given below:
   (a) If possible, find a test case that does **not reach** the mutant.
   (b) If possible, find a test input that satisfies reachability but **not infection** for the mutant.
   (c) If possible, find a test input that satisfies infection, but **not propagation** for the mutant.
   (d) If possible, find a test input that kills mutant m.
   (a) while (isPat == false && isub + patternLen - 1 < subjectLen)
   while (isPat == false && isub + patternLen - 0 < subjectLen) // Mutant A
   (b) isPat = false;
   isPat = true; // Mutant B
4. Why does it make sense to remove ineffective test cases?

5. Define 12 mutants for the following method *cal*() using the effective mutation operators given previously. Try to use each mutation operator at least once. Approximately how many mutants do you think there would be if all mutants for cal were created?

```
public static int cal (int month1, int day1, int month2, int day2, int year)
{
//*********************************************************
// Calculate the number of Days between the two given days in
// the same year.
// preconditions : day1 and day2 must be in same year
//                 1 <= month1, month2 <= 12
//                 1 <= day1, day2 <= 31
//                 month1 <= month2
//                 The range for year: 1 ... 10000
//*********************************************************
  int numDays;

   if (month2 == month1) // in the same month
      numDays  = day2 - day1;
   else
   {
      // Skip month 0.
      int daysIn[] = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
      // Are we in a leap year?
      int m4 = year % 4;
      int m100 = year % 100;
      int m400 = year % 400;
      if ((m4 != 0) || ((m100 ==0) && (m400 != 0)))
         daysIn[2] = 28;
      else
         daysIn[2] = 29;

      // start with days in the two months
      numDays = day2 + (daysIn[month1] - day1);

      // add the days in the intervening months
      for (int i = month1 + 1; i <= month2-1; i++)
         numDays = daysIn[i] + numDays;
   }
   return (numDays);
}
```

6. Define 12 mutants for the following method *power*() using the effective mutation operators given previously. Try to use each mutation operator at least once. Approximately how many mutants do you think there would be if all mutants for *power*() were created?

```
public static int power (int left, int right)
{
//*************************************
// Raises Left to the power of Right
// precondition : Right >= 0
// postcondition: Returns Left**Right
//*************************************
  int rslt;
  rslt = Left;
  if (Right == 0)
  {
    rslt = 1;
  }
  else
  {
    for (int i = 2; i <= Right; i++)
        rslt = rslt * Left;
  }
  return (rslt);
}
```

7. The fundamental premise was stated as: "*In practice, if the software contains a fault, there will usually be a set of mutants that can be killed only by a test case that also detects that fault.*"
   (a) Give a brief argument **in support of** the fundamental mutation premise.
   (b) Give a brief argument **against** the fundamental mutation premise.
8. Try to design mutation operators that subsume combinatorial coverage. Why wouldn't we want such an operator?
9. Look online for the tool Jester, which is based on JUnit.[4] Based on your reading, evaluate Jester as a mutation-testing tool.
10. Download and install the Java mutation tool *muJava*: http://ise.gmu.edu/~offutt/mujava/. Enclose the method *cal*() from the previous question inside a class, and use muJava to test *cal*(). Note that a test case is a method call to *cal*().
    (a) How many mutants are there?
    (b) How many test cases do you need to kill the mutants?
    (c) What mutation score were you able to achieve without analyzing for equivalent mutants?
    (d) How many equivalent mutants are there?

5.3 INTEGRATION AND OBJECT-ORIENTED TESTING

This book uses the term "integration testing" to refer to testing connections among separate program units. In Java, that involves testing the way classes, packages, and components are connected. This section uses the general term "component." This is also where features that are unique to object-oriented programming languages are tested, specifically, inheritance, polymorphism, and dynamic binding.

5.3.1 BNF Integration Testing

As far as we know, BNF testing has not been used at the integration level.

5.3.2 Integration Mutation

This section first discusses how mutation can be used for testing at the integration level without regard to object-oriented relationships, then how mutation can be used to test for problems involving inheritance, polymorphism, and dynamic binding.

Faults that can occur in the integration between two components usually depend on a mismatch of assumptions. For example, Chapter 1 discussed the Mars lander of September 1999, which crashed because a component sent a value in English units (miles) and the recipient component assumed the value was in kilometers. Whether such a flaw should be fixed by changing the caller, the callee, or both depends on the design specification of the program and possibly pragmatic issues such as which is easier to change.

Integration mutation (also called "interface mutation") works by creating mutations on the connections between components. Most mutations are around method calls, and both the calling (caller) and called (callee) method must be considered. Interface mutation operators do the following:

- Change a calling method by modifying the values that are sent to a called method.
- Change a calling method by modifying the call.
- Change a called method by modifying the values that enter and leave a method. This should include parameters as well as variables from a higher scope (class level, package, public, etc.).
- Change a called method by modifying statements that return from the method.

1. ***IPVR – Integration Parameter Variable Replacement***:

Each parameter in a method call is replaced by each other variable of compatible type in the scope of the method call.

   IPVR does not use variables of an incompatible type because they would be syntactically illegal (the compiler should catch them). In OO languages, this operator replaces primitive type variables as well as objects.

2. ***IUOI – Integration Unary Operator Insertion***:

Each expression in a method call is modified by inserting all possible unary operators in front and behind it.

   The unary operators vary by language and type. Java includes ++ and −− as both prefix and postfix operators for numeric types.

3. ***IPEX – Integration Parameter Exchange***:

Each parameter in a method call is exchanged with each parameter of compatible type in that method call.

   For example, if a method call is `max (a, b)`, a mutated method call of `max (b, a)` is created.

4. ***IMCD – Integration Method Call Deletion***:

Each method call is deleted. If the method returns a value and it is used in an expression, the method call is replaced with an appropriate constant value.

In Java, the default values should be used for methods that return values of primitive type. If the method returns an object, the method call should be replaced by a call to **new()** on the appropriate class.

5. ***IREM – Integration Return Expression Modification***:

Each expression in each return statement in a method is modified by applying the UOI and AOR operators from Section 5.2.2.

Object-Oriented Mutation Operators

Chapter 1 defined intra-method, inter-method, intra-class, and inter-class testing. Those mutation operators can be used at the inter-method level (between methods in the same class) and at the inter-class level (between methods in different classes). When testing at the inter-class level, testers also have to worry about faults in the use of inheritance and polymorphism. These are powerful language features that can solve difficult programming problems, but also introduce difficult testing problems.

Languages that include features for inheritance and polymorphism often also include features for information hiding and overloading. Thus, mutation operators to test those features are usually included with the OO operators, even though these are not usually considered to be essential to calling a language "object-oriented."

To understand how mutation testing is applied to such features, we need to examine the language features in depth. This is done in terms of Java; other OO languages tend to be similar but with some subtle differences.

Encapsulation is an abstraction mechanism to enforce information hiding, a design technique that frees clients of an abstraction from unnecessary dependence on design decisions in the implementation of the abstraction. Encapsulation allows objects to restrict access to their member variables and methods by other objects. Java supports four distinct access levels for member variables and methods: private, protected, public, and default (also called package). These access levels are poorly understood by many programmers, and often not carefully considered during design, so they are a rich source of faults. Table 5.1 summarizes these access levels. A *private* member is available only to the class in which it is defined. If access is not specified, the access level defaults to *package*, which allows access to classes in the same package, but **not** subclasses in other packages. A *protected* member is available

Table 5.1. Java's access levels

Specifier	Same class	Different class/ same package	Different package subclass	Different package non-subclass
private	Y	n	n	n
package	Y	Y	n	n
protected	Y	Y	Y	n
public	Y	Y	Y	Y

to the class itself, subclasses, and classes in the same package. A *public* member is available to any class in any inheritance hierarchy or package (the world).

Java does not support multiple class inheritance, so every class has only one immediate parent. A subclass inherits variables and methods from its parent and all of its ancestors, and can use them as defined, or override the methods or hide the variables. Subclasses can also explicitly use their parent's variables and methods using the keyword "super" (`super.methodname();`). Java's inheritance allows method overriding, variable hiding, and class constructors.

Method overriding allows a method in a subclass to have the same name, arguments and result type as a method in its parent. Overriding allows subclasses to redefine inherited methods. The child class method has the same signature, but a different implementation.

Variable hiding is achieved by defining a variable in a child class that has the same name and type of an inherited variable. This has the effect of hiding the inherited variable from the child class. This is a powerful feature, but it is also a potential source of errors.

Class constructors are not inherited in the same way other methods are. To use a constructor of the parent, we must explicitly call it using the *super* keyword. The call must be the first statement in the derived class constructor and the parameter list must match the parameters in the argument list of the parent constructor.

Java supports two versions of polymorphism, attributes and methods, both of which use dynamic binding. Each object has a *declared type* (the type in the declaration statement, that is, "*Parent P;*") and an *actual type* (the type in the instantiation statement, that is, "*P = new Child*();," or the assignment statement, "*P = Pold;*"). The actual type can be the declared type or any type that is descended from the declared type.

A *polymorphic attribute* is an object reference that can take on various types. At any location in the program, the type of the object reference can be different in different executions. A *polymorphic method* can accept parameters of different types by having a parameter that is declared of type *Object*. Polymorphic methods are used to implement *type abstraction* (templates in C++ and generics in Ada).

Overloading is the use of the same name for different constructors or methods in the same class. They must have different *signatures*, or lists of arguments. Overloading is easily confused with overriding because the two mechanisms have similar names and semantics. Overloading occurs with two methods in the same class, whereas overriding occurs between a class and one of its descendants.

In Java, member variables and methods can be associated with the class rather than with individual objects. Members associated with a class are called *class* or *static variables* and *methods*. The Java run time system creates a single copy of a static variable the first time it encounters the class in which the variable is defined. All instances of that class share the same copy of the static variable. Static methods can operate only on static variables; they cannot access instance variables defined in the class. Unfortunately the terminology varies; we say *instance variables* are declared at the class level and are available to objects, *class variables* are declared with static, and *local variables* are declared within methods.

Mutation operators can be defined for all of these language features. The purpose of mutating them is to make sure that the programmer is using them correctly. One reason to be particularly concerned about the use of OO language

features is because many programmers today have learned them "on the job," without having the opportunity to study the theoretical rules about how to use them appropriately.

Following are mutation operators for information hiding language features, inheritance, polymorphism and dynamic binding, method overloading, and classes.

1. ***AMC – Access Modifier Change***:

The access level for each instance variable and method is changed to other access levels.

   The AMC operator helps testers generate tests that ensure that accessibility is correct. These mutants can be killed only if the new access level denies access to another class or allows access that causes a name conflict.

2. ***HVD – Hiding Variable Deletion***:

Each declaration of an overriding, or hiding variable is deleted.

   This causes references to that variable to access the variable defined in the parent (or ancestor), which is a common programming mistake.

3. ***HVI – Hiding Variable Insertion***:

A declaration is added to hide the declaration of each variable declared in an ancestor.

   These mutants can be killed only by test cases that can show that the reference to the overriding variable is incorrect.

4. ***OMD – Overriding Method Deletion***:

Each entire declaration of an overriding method is deleted.

   References to the method will then use the parent's version. This ensures that the method invocation is to the intended method.

5. ***OMM – Overridden Method Moving***:

Each call to an overridden method is moved to the first and last statements of the method and up and down one statement.

   Overriding methods in child classes often call the original method in the parent class, for example to modify a variable that is private to the parent. A common mistake to make is to call the parent's version at the wrong time, which can cause incorrect state behavior.

6. ***OMR – Overridden Method Rename***:

Renames the parent's versions of methods that are overridden in a subclass so that the overriding does not affect the parent's method.

   The OMR operator is designed to check whether an overriding method causes problems with other methods. Consider a method *m*() that calls another method *f*(), both in a class *List*. Further, assume that *m*() is inherited without change in a child class *Stack*, but *f*() is overridden in *Stack*. When *m*() is called on an object of type *Stack*, it calls *Stack*'s version of *f*() instead of *List*'s version. In this case, *Stack*'s version of *f*() may have an interaction with the parent's version that has unintended consequences.

7. ***SKD – Super Keyword Deletion***:

Delete each occurrence of the *super* keyword.

After the change, the reference will be to the local version instead of the ancestor's version. The SKD operator is designed to ensure that hiding/hidden variables and overriding/overridden methods are used appropriately.

8. ***PCD – Parent Constructor Deletion***:

Each call to a *super* constructor is deleted.

The parent's (or ancestor's) default constructor will be used. To kill these mutants, it is necessary to find a test case for which the parent's default constructor creates an initial state that is incorrect.

9. ***ATC – Actual Type Change***:

The actual type of a new object is changed in the `new()` statement.

This causes the object reference to refer to an object of a type that is different from the original actual type. The new actual type must be in the same "type family" (a descendant) of the original actual type.

10. ***DTC – Declared Type Change***:

The declared type of each new object is changed in the declaration.

The new declared type must be an ancestor of the original type. The instantiation will still be valid (it will still be a descendant of the new declared type). To kill these mutants, a test case must cause the behavior of the object to be incorrect with the new declared type.

11. ***PTC – Parameter Type Change***:

The declared type of each parameter object is changed in the declaration.

This is the same as DTC except on parameters.

12. ***RTC – Reference Type Change***:

The right side objects of assignment statements are changed to refer to objects of a compatible type.

For example, if an `Integer` is assigned to a reference of type `Object`, the assignment may be changed to that of a `String`. Since both Integers and Strings are descended from Object, both can be assigned interchangeably.

13. ***OMC – Overloading Method Change***:

For each pair of methods that have the same name, the bodies are interchanged.

This ensure that overloaded methods are invoked appropriately.

14. ***OMD – Overloading Method Deletion***:

Each overloaded method declaration is deleted, one at a time.

The OMD operator ensures coverage of overloaded methods; that is, all the overloaded methods must be invoked at least once. If the mutant still

works correctly without the deleted method, there may be an error in invoking one of the overloading methods; the incorrect method may be invoked or an incorrect parameter type conversion has occurred.

15. ***AOC – Argument Order Change***:

The order of the arguments in method invocations is changed to be the same as that of another overloading method, if one exists.

This causes a different method to be called, thus checking for a common fault in the use of overloading.

16. ***ANC – Argument Number Change***:

The number of the arguments in method invocations is changed to be the same as that of another overloading method, if one exists.

This helps ensure that the programmer did not invoke the wrong method. When new values need to be added, they are the constant default values of primitive types or the result of the default constructors for objects.

17. ***TKD* – this *Keyword Deletion***:

Each occurrence of the keyword *this* is deleted.

Within a method body, uses of the keyword *this* refers to the current object if the member variable is hidden by a local variable or method parameter that has the same name. The TKD operator checks if the member variables are used correctly by replacing occurrences of "*this.X*" with "*X*."

18. ***SMC* – Static *Modifier Change***:

Each instance of the *static* modifier is removed, and the *static* modifier is added to instance variables.

This operator validates use of instance and class variables.

19. ***VID – Variable Initialization Deletion***:

Remove initialization of each member variable.

Instance variables can be initialized in the variable declaration and in constructors for the class. The VID operator removes the initializations so that member variables are initialized to the default values.

20. ***DCD – Default Constructor Delete***:

Delete each declaration of default constructor (with no parameters).

This ensures that user-defined default constructors are implemented properly.

5.4 SPECIFICATION-BASED GRAMMARS

The general term "specification-based" is applied to languages that describe software in abstract terms. This includes formal specification languages such as Z, SMV, OCL, and informal specification languages and design notations such as statecharts, FSMs, and other UML diagram notations. Such languages are becoming more widely used, partly because of increased emphasis on software quality and partly because of the widespread use of the UML.

5.4.1 BNF Grammars

To our knowledge, terminal symbol coverage and production coverage have been applied to only one type of specification language: algebraic specifications. The idea is to treat an equation in an algebraic specification as a production rule in a grammar and then derive strings of method calls to cover the equations. As algebraic specifications are not widely used, this book does not discuss this topic.

5.4.2 Specification-Based Mutation

Mutation testing can also be a valuable method at the specification level. In fact, for certain types of specifications, mutation analysis is actually easier. We address specifications expressed as finite state machines in this section.

A finite state machine is essentially a graph G, as defined in Chapter 2, with a set of states (nodes), a set of initial states (initial nodes), and a transition relation (the set of edges). When finite state machines are used, sometimes the edges and nodes are explicitly identified, as in the typical bubble and arrow diagram. However, sometimes the finite state machine is more compactly described in the following way.

1. States are implicitly defined by declaring variables with limited ranges. The state space is then the Cartesian product of the ranges of the variables.
2. Initial states are defined by limiting the ranges of some or all of the variables.
3. Transitions are defined by rules that characterize the source and target of each transition.

The following example clarifies these ideas. We describe a machine with a simple syntax, and show the same machine with explicit enumerations of the states and transitions. Although this example is too small to show this point, the syntax version in SMV is typically *much* smaller than the graph version. In fact, since state space growth is combinatorial, it is quite easy to define finite state machines where the explicit version is far too long to write, even though the machine itself can be analyzed efficiently. Below is an example in the SMV language.

```
MODULE main
#define false 0
#define true 1

VAR
    x, y : boolean;

ASSIGN
    init (x) := false;
    init (y) := false;

    next (x) := case
      !x & y : true;
      !y     : true;
      x      : false;
```

```
  true   : x;
esac;

next (y) := case
  x & !y : false;
  x & y  : y;
  !x & y : false;
  true   : true;
esac;
```

Two variables appear, each of which can have only two values (boolean), so the state space is of size $2 * 2 = 4$. One initial state is defined, in the two init statements under ASSIGN. The transition diagram is shown in Figure 5.4. Transition diagrams for SMV can be derived by mechanically following the specifications. Take a given state and decide what the next value for each variable is. For example, assume the above specification is in the state (*true*, *true*). The next value for x will be determined by the "x : false" statement. x is *true*, so its next value will be *false*. Likewise, x & y is true, so the next value of y will be its current value, or *true*. Thus, the state following (*true*, *true*) is (*false*, *true*). If multiple conditions in a case statement are true, the first one that is true is chosen. SMV has no "fall-through" semantics, such as one might find in a language such as C or Java.

Our context has two particularly important aspects of such a structure.

1. Finite state descriptions can capture system behavior at a very high level – suitable for communicating with the end user. Finite state machines are incredibly useful for the hardest part of testing, namely system testing.
2. The verification community has built powerful analysis tools for finite state machines. These tools are highly automated. Further, these tools produce explicit evidence, in the form of witnesses or counterexamples, for properties that do not hold in the finite state machine. These counterexamples can be interpreted as test cases. Thus, it is easier to automate test case generation from finite state machines than from program source.

Mutations and Test Cases

Mutating the syntax of state machine descriptions is very much like mutating program source. Mutation operators must be defined, and then they are applied to

Figure 5.4. Finite state machine for SMV specification.

Figure 5.5. Mutated finite state machine for SMV specification.

the description. One example is the *Constant Replacement* operator, which replaces each constant with other constants. Given the phrase `!x & y : false` in the `next` statement for `y`, replace it with `!x & y : true`. The finite state machine for this mutant is shown in Figure 5.5. The new transition is drawn as an extra thick arrow and the replaced transition is shown as a crossed-out dotted arrow.

Generating a test case to kill this mutant is a little different from program-based mutation. We need a sequence of states that is allowed by the transition relation of the original state machine, but not by the mutated state machine. Such a sequence is precisely a test case that kills the mutant.

Finding a test to kill a mutant of a finite state machine expressed in SMV can be automated using a *model checker*. A model checker takes two inputs. The first is a finite state machine, described in a formal language such as SMV. The second is a statement of some property, expressed in a *temporal logic*. We will not fully explain temporal logic here, other than to say that such a logic can be used to express only properties that are true "now," and also properties that will (or might) be true in the future. The following is a simple temporal logic statement:

> The original expression, `!x & y : false` in this case, is **always** the same as the mutated expression, `x | y : true`.

For the given example, this statement is false with respect to a sequence of states allowed by the original machine if and only if that sequence of states is rejected by the mutant machine. In other words, such a sequence in question is a test case that kills the mutant. If we add the following SMV statement to the above machine

```
SPEC AG (!x & y) ⟶ (y = false)
```

The model checker will obligingly produce the desired test sequence:

```
/* state 1 */ { x = 0, y = 0 }
/* state 2 */ { x = 1, y = 1 }
/* state 3 */ { x = 0, y = 1 }
/* state 4 */ { x = 1, y = 0 }
```

Some mutated state machines are equivalent to the original machine. The model checker is exceptionally well adapted to deal with this. The key theoretical reason is that the model checker has a finite domain to work in, and hence the equivalent mutant problem is decidable (unlike with program code). In other words, if the model checker does not produce a counterexample, we *know* that the mutant is equivalent.

EXERCISES

Section 5.4.

1. (Challenging!) Find or write a small SMV specification and a corresponding Java implementation. Restate the program logic in SPEC assertions. Mutate the assertions systematically, and collect the traces from (nonequivalent) mutants. Use these traces to test the implementation.

5.5 INPUT SPACE GRAMMARS

One common use of grammars is to define the syntax of the inputs to a program, method, or software component formally. This section explains how to apply the criteria of this chapter to grammars that define the input space of a piece of software.

5.5.1 BNF Grammars

Section 5.1.1 of this chapter presented criteria on BNF grammars. One common use of a grammar is to define a precise syntax for the input of a program or method.

Consider a program that processes a sequence of deposits and debits, where each deposit is of the form deposit *account amount* and each debit is of the form debit *account amount*. The input structure of this program can be described with the regular expression:

(deposit *account amount*|debit *account amount*)*

This regular expression describes any sequence of deposits and debits. (The example in Section 5.1.1 is actually an abstract version of this example.)

The regular expression input description is still fairly abstract, in that it does not say anything about what an *account* or an *amount* looks like. We will refine those details later. One input that can be derived from this grammar is

```
deposit 739 $ 12.35
deposit 644 $ 12.35
debit 739 $ 19.22
```

It is easy to build a graph that captures the effect of regular expressions. Formally, these graphs are finite automata, either deterministic or nondeterministic. In either case, one can apply the coverage criteria from Chapter 2 directly.

Figure 5.6. Finite state machine for bank example.

One possible graph for the above structure is shown in Figure 5.6. It contains one state (Ready) and two transitions that represent the two possible inputs. The input test example given above satisfies both the all nodes and all edges criteria for this graph.

Although regular expressions suffice for some programs, others require grammars. As grammars are more expressive than regular expressions we do not need to use both. The prior example specified in grammar form, with all of the details for *account* and *amount*, is

```
bank    ::= action*
action  ::= dep | deb
dep     ::= "deposit" account amount
deb     ::= "debit" account amount
account ::= digit^3
amount  ::= "$" digit+ "." digit^2
digit   ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

The graph for even this simple example is substantially larger once all details have been included. It is shown in Figure 5.7.

Figure 5.7. Finite state machine for bank example grammar.

A full derivation of the test case above begins as follows:

```
stream → action^*
       → action action^*
       → dep action^*
       → deposit account amount action^*
       → deposit digit^3 amount action^*
```

→ deposit **digit digit^2** amount action^*
→ deposit **7** digit^2 amount action^*
→ deposit 7 **digit digit** amount action^*
→ deposit 7**3** digit amount action^*
→ deposit 73**9** amount action^*
→ deposit 739 **$ digit^+ . digit^2** action^*
→ deposit 739 $ **digit^2 .** digit^2 action^*
→ deposit 739 $ **digit digit .** digit^2 action^*
→ deposit 739 $**1** digit . digit^2 action^*
→ deposit 739 $1**2.** digit^2 action^*
→ deposit 739 $12**. digit digit** action^*
→ deposit 739 $12.**3 digit** action^*
→ deposit 739 $12.3**5** action^*
⋮

Deriving tests from this grammar proceeds by systematically replacing the next nonterminal (action) with one of its productions. The exercises below ask for complete tests to satisfy terminal symbol coverage and production coverage.

Of course, it often happens that an informal description of the input syntax is available, but not a formal grammar. This means that the test engineer is left with the engineering task of formally describing the input syntax. This process is **extremely** valuable and will often expose ambiguities and omissions in the requirements and software. Thus, this step should be carried out early in development, definitely before implementation and preferably before design. Once defined, it is sometimes helpful to use the grammar directly in the program for execution-time input validation.

XML Example

A language for describing inputs that is quickly gaining popularity is the *eXtensible Markup Language (XML)*. The most common use of XML is in web applications and web services, but XML's structure is generic enough to be useful in many contexts. XML is a language for describing, encoding and transmitting data. All XML "messages" (also sometimes called "documents") are in plain text and use a syntax similar to HTML. XML comes with a built-in language for describing the input messages in the form of a grammar, called *schemas*.

Like HTML, XML uses *tags*, which are textual descriptions of data enclosed in angle brackets ('<' and '>'). All XML messages must be *well formed*, that is, have a single document element with other elements properly nested under it, and every tag must have a corresponding closing tag. A simple example XML message for books is shown in Figure 5.8. This example is used to illustrate the use of BNF testing on software that uses XML messages. The example lists two books. The tag names ("books," "book," "ISBN," etc.) should be self descriptive and the XML message forms an overall hierarchy.

XML documents can be constrained by grammar definitions written in *XML Schemas*. Figure 5.9 shows a schema for books. The schema says that a "books"

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML file for books-->
<books xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="C:\Books\books.xsd">
 <book>
   <ISBN>0471043281</ISBN>
   <title>The Art of Software Testing</title>
   <author>Glen Myers</author>
   <publisher>Wiley</publisher>
   <price>50.00</price>
   <year>1979</year>
 </book>
 <book>
   <ISBN>0442206720</ISBN>
   <title>Software Testing Techniques</title>
   <author>Boris Beizer</author>
   <publisher>Van Nostrand Reinhold, Inc</publisher>
   <price>75.00</price>
   <year>1990</year>
 </book>
</books>
```

Figure 5.8. Simple XML message for books.

XML message can contain an unbounded number of "book" tags. The "book" tags contain six pieces of information. Three, "title," "author," and "publisher," are simple strings. One, "price," is of type decimal (numeric), has two digits after the decimal point and the lowest value is 0. Two data elements, "ISBN" and "year," are types that are defined later in the schema. The type "yearType" is an integer with four digits, and "isbnType" can have up to 10 numeric characters. Each book must have a title, author, publisher, price, and year, and ISBN is optional.

Given an XML schema, the criteria defined in Section 5.1.1 can be used to derive XML messages that serve as test inputs. Following the production coverage criteria would result in two XML messages for this simple schema, one that includes an ISBN and one that does not.

5.5.2 Mutation for Input Grammars

It is quite common to require a program to reject malformed inputs, and this property should definitely be tested. It is the kind of thing that slips past the attention of implementors who are focused on making a program do what it is supposed to do.

Do invalid inputs really matter? From the perspective of program correctness, invalid inputs are simply those outside the precondition of a specified function. Formally speaking, a software implementation of that function can exhibit any behavior on inputs that do not satisfy the precondition. This includes failure to terminate, run time exceptions, and "bus error, core dumps."

However, the correctness of the intended functionality is only part of the story. From a practical perspective, invalid inputs sometimes matter a great deal because they hold the key to unintended functionality. For example, unhandled invalid inputs often represent security vulnerabilities, allowing a malicious party to break the software. Invalid inputs often cause the software to behave in surprising ways, which malicious parties can use to their advantage. This is how the classic "buffer overflow attack" works. The key step in a buffer overflow attack is to provide an

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
 <xs:element name="books">
  <xs:annotation>
   <xs:documentation>XML Schema for Books</xs:documentation>
  </xs:annotation>
  <xs:complexType>
   <xs:sequence>
    <xs:element name="book" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="ISBN" type="xs:isbnType" minOccurs="0"/>
       <xs:element name="title" type="xs:string"/>
       <xs:element name="author" type="xs:string"/>
       <xs:element name="publisher" type="xs:string"/>
       <xs:element name="price" type="xs:decimal" fractionDigits="2" minInclusive="0"/>
       <xs:element name="year" type="yearType"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:simpleType name="yearType">
  <xs:restriction base="xs:int">
   <xs:totalDigits value="4"/>
  </xs:restriction>
 </xs:simpleType>
 <xs:simpleType name="isbnType">
  <xs:restriction base="xs:string">
   <xs:pattern value="[0-9]{10}"/>
  </xs:restriction>
 </xs:simpleType>
</xs:schema>
```

Figure 5.9. XML schema for books.

input that is too long to fit into the available buffer. Similarly, a key step in certain web browser attacks is to provide a string input that contains malicious HTML, Javascript, or SQL. Software should behave "reasonably" with invalid inputs. "Reasonable" behavior may not always be defined, but the test engineer is obliged to consider it anyway.

To support security as well as to evaluate the software's behavior, it is useful to produce test cases that contain invalid inputs. A common way to do this is to mutate a grammar. When mutating grammars, the mutants are the tests and we create valid and invalid strings. No ground string is used, so the notion of killing mutants does not apply to mutating grammars. Several mutation operators for grammars are defined below.

1. ***Nonterminal Replacement***:

Every nonterminal symbol in a production is replaced by other nonterminal symbols.

This is a very broad mutation operator that could result in many strings that are not only invalid, they are so far away from valid strings that they are useless for testing. If the grammar provides specific rules or syntactic restrictions, some nonterminal replacements can be avoided. This is analogous to avoiding compiler errors in program-based mutation. For example, some strings represent type structures and only nonterminals of the same or compatible type should be replaced.

The production `dep ::= "deposit" account amount` can be mutated to create the following productions:

```
dep ::= "deposit" amount amount
dep ::= "deposit" account digit
```

Which can result in the corresponding tests:

```
debit 739 $12.35
deposit $19.22 $12.35
deposit 739 1
```

2. ***Terminal Replacement***:

Every terminal symbol in a production is replaced by other terminal symbols.

Just as with nonterminal replacement, some terminal replacements may not be appropriate. Recognizing them depends on the particular grammar that is being mutated. For example, the production `amount ::= "$" digit`$^+$ `"."` `digit`^{2} can be mutated to create the following three productions:

```
amount ::= "." digit+ "." digit2
amount ::= "$" digit+ "$" digit2
amount ::= "$" digit+ "1" digit2
```

Which can result in the corresponding tests:

```
deposit 739 .12.35
deposit 739 $12$35
deposit 739 $12135
```

3. ***Terminal and Nonterminal Deletion***:

Every terminal and nonterminal symbol in a production is deleted.

For example, the production `dep ::= "deposit" account amount` can be mutated to create the following three productions:

```
dep ::= account amount
dep ::= "deposit" amount
dep ::= "deposit" account
```

Which can result in the corresponding tests:

```
739 $12.35
deposit $12.35
deposit 739
```

4. ***Terminal and Nonterminal Duplication***:

Every terminal and nonterminal symbol in a production is duplicated.

This is sometimes called the "stutter" operator. For example, the production `dep ::= "deposit" account amount` can be mutated to create the following three mutated productions:

```
dep ::= "deposit" "deposit" account amount
dep ::= "deposit" account account amount
dep ::= "deposit" account amount amount
```

Which can result in the corresponding tests:

```
deposit deposit 739 $12.35
deposit 739 739 $12.35
deposit 739 $12.35 $12.35
```

We have significantly more experience with program-based mutation operators than grammar-based operators, so this list should be treated as being much less definitive.

These mutation operators can be applied in either of two ways. One is to mutate the grammar and then generate inputs. The other is to use the correct grammar, but one time during each derivation apply a mutation operator to the production being used. The operators are typically applied during production, because the resulting inputs are usually "closer" to valid inputs than if the entire grammar is corrupted. This approach is used in the previous examples.

Just as with program-based mutation, some inputs from a mutated grammar rule are still in the grammar. The example above of changing the rule

```
dep    ::= "deposit" account amount
```

to be

```
dep    ::= "debit" account amount
```

yields an "equivalent" mutant. The resulting input, `debit 739 $12.35`, is a valid input, although the effects are (sadly) quite different for the customer. If the idea is to generate invalid inputs exclusively, some way must be found to screen out mutant inputs that are valid. Although this sounds much like the equivalence problem for programs, the difference is small but significant. Here the problem is solvable and can be solved by creating a recognizer from the grammar, and checking each string as it is produced.

Many programs are supposed to accept some, but not all, inputs from some larger language. For example, a Web application might restrict its inputs to a subset of HTML. In this case, we have two grammars: the full grammar, and a grammar for the subset. In this case, the most useful invalid tests to generate are those that are in the first grammar, but not in the second.

XML Example

Section 5.5.1 showed examples of generating tests in the form of XML messages from a schema grammar definition. It is also convenient to apply mutation to XML schemas to produce invalid messages. Some programs will use XML parsers that

validate the messages against the grammar. If they do, it is likely that the software will usually behave correctly on invalid messages, but testers still need to verify this. If a validating parser is not used, this can be a rich source for programming mistakes. It is also fairly common for programs to use XML messages without having an explicit schema definition. In this case, it is very helpful for the test engineer to develop the schema as a first step to developing tests.

XML schemas have a rich collection of built-in datatypes, which come with a large number of *constraining facets*. In XML, *constraining facets* are used to restrict further the range of values. The example in Figure 5.9 uses several constraining facets, including *fractionDigits*, *minInclusive*, and *minOccurs*. This suggests further mutation operators for XML schemas that modify the **values** of facets. This can often result in a rich collection of tests for software that use inputs described with XML.

Given the following four lines in the books schema in Figure 5.9:

```
<xs:element name="ISBN" type="xs:isbnType" minOccurs="0"/>
<xs:element name="price" type="xs:decimal" fractionDigits="2" minInclusive="0"/>
<xs:totalDigits value="4"/>
<xs:pattern value="[0-9]{10}"/>
```

we might construct the mutants:

```
<xs:element name="ISBN" type="xs:isbnType" minOccurs="1"/>

<xs:element name="price" type="xs:decimal" fractionDigits="1" minInclusive="0"/>
<xs:element name="price" type="xs:decimal" fractionDigits="3" minInclusive="0"/>
<xs:element name="price" type="xs:decimal" fractionDigits="2" minInclusive="1"/>
<xs:element name="price" type="xs:decimal" fractionDigits="2" maxInclusive="0"/>

<xs:totalDigits value="5"/>
<xs:totalDigits value="0"/>

<xs:pattern value="[0-8]{10}"/>
<xs:pattern value="[1-9]{10}"/>
<xs:pattern value="[0-9]{9}"/>
```

EXERCISES

Section 5.5.

1. Generate tests to satisfy TSC for the bank example grammar based on the BNF in Section 5.1.1. Try **not** to satisfy PDC.
2. Generate tests to satisfy PDC for the bank example grammar.
3. Consider the following BNF with start symbol A:

```
A ::= B"@"C"."B
B ::= BL | L
C ::= B | B"."B
L ::= "a" | "b" | "c" | ... | "y" | "z"
```

and the following six possible test cases:

```
t1 = a@a.a
t2 = aa.bb@cc.dd
t3 = mm@pp
t4 = aaa@bb.cc.dd
t5 = bill
t6 = @x.y
```

For each of the six tests, (1) identify the test sequence as either "in" the BNF, and give a derivation, or (2) identify the test sequence as "out" of the BNF, and give a mutant derivation that results in that test. (Use only one mutation per test, and use it only one time per test).

4. Provide a BNF description of the inputs to the *cal()* method in the homework set for Section 5.2.2. Succinctly describe any requirements or constraints on the inputs that are hard to model with the BNF.
5. Answer questions (a) through (c) for the following grammar:

```
val    ::= number | val pair
number ::= digit+
pair   ::= number op | number pair op
op     ::= "+" | "-" | "*" | "/"
digit  ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Also consider the following mutation, which adds an additional rule to the grammar:

```
pair ::= number op | number pair op | op number
```

(a) Which of the following strings can be generated by the (unmutated) grammar?

```
42
4 2
4 + 2
4 2 +
4 2 7 - *
4 2 - 7 *
4 2 - 7 * +
```

(b) Find a string that is generated by the mutated grammar, but not by the original grammar.

(c) (*Challenging*) Find a string whose generation uses the new rule in the mutant grammar, but is also in the original grammar. Demonstrate your answer by giving the two relevant derivations.

6. Answer questions (a) and (b) for the following grammar.

```
phoneNumber ::= exhangePart dash numberPart
exchangePart ::= special zeroOrSpecial other
numberPart   ::= ordinary4
ordinary     ::= zero | special | other
zeroOrSpecial ::= zero | special
```

```
zero        ::= "0"
special     ::= "1" | "2"
other       ::= "3" | "4" | "5" | "6" | "7" | "8" | "9"
dash        ::= "-"
```

(a) Classify the following as either phoneNumbers (or not). For non-phone numbers, indicate the problem.

- 123-4567
- 012-3456
- 109-1212
- 346-9900
- 113-1111

(b) *Consider the following mutation of the grammar*:

```
exchangePart ::= special  ordinary  other
```

If possible, identify a string that appears in the mutated grammar but not in the original grammar, another string that is in the original but not the mutated, and a third string that is in both.

7. Java provides a package, *java.util.regex*, to manipulate regular expressions. Write a regular expression for URLs and then evaluate a set of URLs against your regular expression. This assignment involves programming, since input structure testing without automation is pointless.
   (a) Write (or find) a regular expression for a URL. Your regular expression does not need to be so general that it accounts for every possible URL, but give your best effort (for example "*" will not be considered a good effort). You are strongly encouraged to do some web surfing to find some candidate regular expressions. One suggestion is to visit the *Regular Expression Library*.
   (b) Collect a set of URLs from a small web site (such as a set of course web pages). Your set needs to contain at least 20 (different) URLs. Use the *java.util.regex* package to validate each URL against your regular expression.
   (c) Construct a valid URL that is not valid with respect to your regular expression (and show this with the appropriate *java.util.regex call*). If you have done an outstanding job in part 1, explain why your regular expression does not have any such URLs.
8. Why is the equivalent mutant problem solvable for BNF grammars but not for program-based mutation? (Hint: The answer to this question is based on some fairly subtle theory.)

5.6 BIBLIOGRAPHIC NOTES

We trace the use of grammars for testing compilers back to Hanford [150], who motivated subsequent related work [26, 107, 176, 285, 294]. Maurer's Data Generation Language (DGL) tool [231] showed the applicability of grammar-based generation to many types of software, a theme echoed in detail by Beizer [29].

Legend has it that the first ideas of mutation analysis were postulated in 1971 in a class term paper by Richard Lipton. The first research papers were published by

Budd and Sayward [52], Hamlet [148], and DeMillo, Lipton, and Sayward [99] in the late 1970s; DeMillo, Lipton, and Sayward's paper [99] is generally cited as the seminal reference. Mutation has primarily been applied to software by creating mutant versions of the source, but has also been applied to formal software specifications.

The original analysis of the number of mutants was by Budd [53], who analyzed the number of mutants generated for a program and found it to be roughly proportional to the product of the number of variable references times the number of data objects ($O(Refs*Vars)$). A later analysis [5] claimed that the number of mutants is $O(Lines*Refs)$–assuming that the number of data objects in a program is proportional to the number of lines. This was reduced to $O(Lines*Lines)$ for most programs; this figure appears in most of the literature.

A statistical regression analysis of actual programs by Offutt et al. [269] showed that the number of lines did **not** contribute to the number of mutants, but that Budd's figure is accurate. The selective mutation approach mentioned under "Designing Mutation Operators" eliminates the number of data objects so that the number of mutants is proportional to the number of variable references ($O(Refs)$).

A variant of mutation that has been widely discussed is weak mutation [134, 167, 358, 271]. However, experimentation has shown that the difference is very small [163, 226, 271]. Mutation operators have been designed for various programming languages, including Fortran IV [19, 56], COBOL [151], Fortran 77 [101, 187], C [95], C integration testing [94], Lisp [55], Ada [40, 276], Java [185], and Java class relationships [219, 220].

Research proof-of-concept tools have been built for Fortran IV and 77, COBOL, C, Java, and Java class relationships. By far the most widely used tool is Mothra, a mutation system for Fortran 77 that was built in the mid 1980s at Georgia Tech. Mothra was built under the leadership of Rich DeMillo, with most of the design done by DeMillo and Offutt, and most of the implementation by Offutt and King, with help from Krauser and Spafford. In its heyday in the early 1990s, Mothra was installed at well over a hundred sites and the research that was done to build Mothra and that later used Mothra as a laboratory resulted in around half a dozen PhD dissertations and many dozens of papers. As far as we know, the only commercial tool that supports mutation is by the company Certess, in the chip design industry.

The coupling effect says that complex faults are coupled to simple faults in such a way that test data that detects all simple faults will detect most complex faults [99]. The coupling effect was supported empirically for programs in 1992 [263], and has shown to hold probabilistically for large classes of programs in 1995 [335]. Budd [51] discussed the concept of program neighborhoods. The neighborhood concept was used to present the competent programmer hypothesis [99]. The fundamental premise of mutation testing, as coined by Geist et al. [133] is: **in practice, if the software contains a fault, there will usually be a set of mutants that can be killed only by a test case that also detects that fault**.

The operation of replacing each statement with a "bomb" was called Statement ANalysis (SAN) in Mothra [187]. Mothra's Relational Operator Replacement (ROR) operator replaces each occurrence of a relational operator ($<, >, \leq, \geq, =, \neq$) with each other operator and the expression with *true* and *false*. The above subsumption proofs used only the latter operators. Mothra's Logical Connector Replacement (LCR) operator replaces each occurrence of one of the logical operators ($\wedge, \vee, \equiv, \neq$)

with each other operator and the entire expression with *true*, *false*, *leftop* and *rightop*. *leftop* and *rightop* are special mutation operators that return the left side and the right side, respectively, of a relational expression. The mutation operator that removes each statement in the program is called Statement DeLetion (SDL) in Mothra [187].

A number of authors [14, 15, 37, 298, 350] have used traces from model checkers to generate tests, including mutation based tests. The text from Huth and Ryan [173] provides a easily accessible introduction to model checking and discusses use of the SMV system.

One of the key technologies being used to transmit data among heterogeneous software components on the Web is the eXtensible Markup Language (XML) [1, 43]. Data-based mutation defines **generic classes** of mutation operators. These mutation operator classes are intended to work with different grammars. The current literature [203] cites operator classes that modify the length of value strings and determine whether or not a value is in a predefined set of values.

NOTES

1 There is no relationship between this use of mutation and genetic algorithms, except that both make an analogy to biological mutation. Mutation for testing predated genetic algorithms by a number of years.
2 Of course, since mutant detection is undecidable, a heuristic is the best option possible.
3 This goal is in some sense equivalent to the goal of forcing each clause in each predicate to make a difference.
4 Jester's web page is http://jester.sourceforge.net/

PART 3

Applying Criteria in Practice

6

Practical Considerations

The first five chapters of this book help practical testers fill up their "toolbox" with various test criteria. While studying test criteria in isolation is needed, one of the most difficult obstacles to a software organization moving to level 3 or 4 testing is integrating effective testing strategies into the overall software development process. This chapter discusses various issues involved with applying test criteria during software development. The overriding philosophy is *to think*: if the tester looks at this as a technical problem, and seeks technical solutions, he or she will usually find that the obstacles are not as large as they first seem.

6.1 REGRESSION TESTING

Regression testing is the process of re-testing software that has been modified. Regression testing constitutes the vast majority of testing effort in commercial software development and is an essential part of any viable software development process. Large components or systems tend to have large regression test suites. Even though many developers don't want to believe it (even when faced with indisputable evidence!), small changes to one part of a system often cause problems in distant parts of the system. Regression testing is used to find this kind of problem.

It is worth emphasizing that regression tests *must* be automated. Indeed, it could be said that unautomated regression testing is equivalent to no regression testing. A wide variety of commercially available tools are available. Capture/replay tools automate testing of programs that use graphical user interfaces. Version control software, already in use to manage different versions of a given system, effectively manages the test sets associated with each version. Scripting software manages the process of obtaining test inputs, executing the software, marshaling the outputs, comparing the actual and expected outputs, and generating test reports.

The aim of this section is to explain what kinds of tests ought to be in a regression test set, which regression tests to run, and how to respond to regression tests that fail. We treat each of these issues in turn. We direct the reader interested in detail on any of these topics to the bibliographic notes.

The test engineer faces a Goldilocks problem in determining which tests to include in a regression test set. Including every test set possible results in an unmanageably large regression test set. The result is that the test set cannot be run as often as changes are made to the software. For many development organizations, this period amounts to a day; regression tests run at night to evaluate software changed that day, with developers reviewing the results the following morning. If the regression tests do not finish in a timely manner, the development process is disrupted. It is well worth throwing money at this problem in terms of additional computational resources to execute the tests, but, at some point, the marginal advantage of adding a given test is not worth the marginal expenditure of the resources needed to execute it. On the other side, a set that is too small will not cover the functionality of the software sufficiently well, and too many faults will make it past the regression test set to the customer.

The prior paragraph does not actually say which tests are in the regression test set, just that the set has to be the right size. Some organizations have a policy that for each problem report that has come in from the field, a regression test exists that, in principle, detects the problem. The idea is that customers are more willing to be saddled with new problems than with the same problem over and over. The above approach has a certain charm from the perspective of traceability, in that each test chosen in this way has a concrete rationale.

The coverage criteria that form the heart of this book provide an excellent basis for evaluating regression test sets. For example, if node coverage in the form of method call coverage shows that some methods are never invoked, then it is a good idea to either decide that the method is dead code with respect that particular application, or include a test that results in a call to the method.

If one or more regression tests fail, the first step is to determine if the change to the software is faulty, or if the regression test set itself is broken. In either case, additional work is required. If no regression tests fail, there is still work to do. The reason is that a regression test set that is satisfactory for a given version of the software is not necessarily satisfactory for a subsequent version. Changes to software are often classified as *corrective*, *perfective*, *adaptive*, and *preventive*. All of these changes require regression testing. Even when the (desired) external functionality of the software does not change, the regression test set still needs to be reanalyzed to see if it is adequate. For example, preventive maintenance may result in wholesale internal restructuring of some components. If the criteria used to select the original regression tests were derived from the structure of the implementation, then it is unlikely that the test set will adequately cover the new implementation.

Evolving a regression test set as the associated software changes is a challenge. Changes to the external interface are particularly painful, since such a change can cause all tests to fail. For example, suppose that a particular input moves from one drop-down menu to another. The result is that the capture/playback aspect of executing each test case needs an update. Or suppose that the new version of the software generates an additional output. All of the expected results are now out of date, and need to be augmented. Clearly, automated support for maintaining test sets is just as crucial as automated support for executing the tests.

Adding a (small) number of tests to a regression test set is usually simple. The marginal cost of each additional test is typically quite small. Cumulatively, however,

the test set can become unwieldy. Removing tests from a regression test set is a dicey proposition. Invariably, a fault will show up in the field that one of the removed tests would have found. Fortunately, the same criteria that guide the construction of a regression test set apply when deciding how to update the regression test set.

A different approach to limiting the amount of time needed to execute regression tests, and a focus of much of the attention in the research literature, is selecting only a subset of the regression tests. For example, if the execution of a given test case does not visit *anything* modified, then the test case has to perform the same both before and after the modification, and hence can be safely omitted. Selection techniques include linear equations, symbolic execution, path analysis, data flow analysis, program dependence graphs, system dependence graphs, modification analysis, firewall definition, cluster identification, slicing, graph walks, and modified entity analysis. For example, as a reader of Chapter 2 might guess, data flow selection techniques choose tests only if they touch new, modified, or deleted DU pairs; other tests are omitted.

A selection technique is *inclusive* to the degree that it includes tests that are "modification-revealing." Unsafe techniques have inclusiveness of less than 100%. A technique is *precise* to the extent that it omits regression tests that are not modification-revealing. A technique is *efficient* to the degree that determine the appropriate subset of the regression test set is less computationally intensive than simply executing the omitted tests. Finally, a technique is *general* to the degree that applies to a wide variety of practical situations. To continue the example, the data flow approach to selecting regression tests is not necessarily either safe or precise, of polynomial complexity in certain program attributes, and requires, obviously, data flow information and program instrumentation at the data flow graph level. The bibliographic notes section contains pointers to further details on this work, including empirical evaluations.

6.2 INTEGRATION AND TESTING

Software is composed of many pieces of varying sizes. Individual programmers are often responsible for testing the lowest level components (classes, modules, and methods). After that, testing must be carried out in collaboration with software integration. Software can be integrated in many ways. This section discusses technical strategies that must be employed during testing. We do not try to catalog all of them from a process point of view.

Integration testing is the testing of incompatibilities and interfaces between otherwise correctly working components. That is, it is the testing needed to integrate subcomponents into a bigger working component. This is emphatically not the same as testing an already integrated component.

This chapter uses the term software "component" in a very broad sense: A *component* is a piece of a program that can be tested independently of the complete program or system. Thus, classes, modules, methods, packages, and even code fragments can be considered to be components.

Integration testing is often done with an incomplete system. The tester may be evaluating how only two of many components in the system work together, may be testing integration aspects before the full system is complete, or may be putting the

system together piece by piece and evaluating how each new component fits with the previously integrated components.

6.2.1 Stubs and Drivers

When testing incomplete portions of software, developers and testers often need extra software components, sometimes called *scaffolding*. The two most common types of scaffolding are known as test stubs and test drivers. A *test stub* is a skeletal or special-purpose implementation of a software module, used to develop or test a component that calls the stub or otherwise depends on it. It replaces a called component. For OO programs, the XP community has developed a version of the stub called the mock. A *mock* is a special-purpose replacement class that includes behavior verification to check that a class under test made the correct calls to the mock. A *test driver* is a software component or test tool that replaces a component that takes care of the control and/or the calling of a software component.

One of the responsibilities of a test stub is to return values to the calling component. These values are seldom the same that the actual component being stubbed would return, or else we would not need the stub, but sometimes they must satisfy certain constraints.

The simplest action for a stub is to assign constant values to the outputs. More sophisticated approaches may be to return random values, values from a table lookup, or to let the user enter return values during execution. Test tools have included automated stubbing capabilities since the 1980s. More sophisticated tools discover methods that need to be stubbed, and ask the tester what kind of behavior the stub should have. Some tools collect instances of objects that have the correct return type and make these available as potential stub returns. As a default, this is a powerful approach that the test engineer need only override as necessary. It is possible to generate stubs automatically from formal specifications of the software components, but we are not aware of this functionality in available tools. Programmers also generate their own stubs when carrying out their own unit or module testing.

The simplest form of driver is a `main()` method for a class. Effective programmers often include a `main()` for every class, containing statements that carry out simple testing of the class. If the class is an ADT, the `main()` test driver will create some objects of the class, add values, retrieve values, and use the other operations on the class. Techniques from previous chapters, like sequencing constraints and state-based testing, can be implemented in the driver.

Test drivers can include hard-coded values or retrieve the values from an external source like the tester or a file. Tools exist to generate test drivers automatically. Both test driver and test stub generators are included in other test tools.

6.2.2 Class Integration Test Order

When integrating multiple components, it is important to decide in which order the classes or subsystems should be integrated and tested. Classes depend on each other in various ways. One class may use methods or variables defined in another, a class may inherit from another, or one class may aggregate objects of another class inside its data objects. If class A uses methods defined in class B, and B is not available,

then we need stubs for those methods to test *A*. Therefore, it makes sense to test *B* first, then when *A* is tested we can use actual objects of *B* instead of stubs.

This is called the *class integration test order problem* (CITO), and the general goal is to integrate and test classes in the order that requires the least stubbing, as creating test stubs is considered to be a major cost of integration testing. If the dependencies among the classes have no cycles, the order of their integration is fairly simple. The classes that do not depend on any other classes are tested first. Then they are integrated with classes that depend only on them, and the new classes are tested. If the classes are represented as nodes in a "dependency graph," with edges representing dependencies, this approach follows a *topological sorting* of the graph.

The problem gets more complicated when the dependency graph has cycles, because we will eventually get to a class that depends on another class that has not yet been integrated and tested. This is when some sort of stubbing is required. For example, assume that class *A* uses methods in class *B*, *B* uses methods in class *C*, and *C* aggregates an object of class *A*. When this happens, the integration tester must "break the cycle" by choosing one class in the cycle to test first. The hope is to choose the class that results in the least extra work (primarily that of creating stubs).

Software designers may observe that class diagrams often have few if any cycles and in fact, most design textbooks strongly recommend against including cycles in designs. However, it is common to add classes and relationships as design progresses, for example, to improve performance or maintainability. As a result, class diagrams usually contain cycles by the end of low-level design or implementation, and practical testers have to solve the CITO problem.

The research literature proposes numerous solutions to the CITO problem. This is still an active research area and these solutions have not yet made it into commercial tools.

6.3 TEST PROCESS

Many organizations postpone all software testing activities to the end of development, after the implementation has started, or even after it has ended. By waiting until this late in the process, testing winds up being compressed, not enough resources (time and budget) remain, problems with previous stages have been solved by taking time and dollars from testing, and testers do not have enough time to plan for testing. Instead of planning and designing tests, the developers have time only to run tests, usually in an ad hoc manner. The key point is that the goal is to create high quality software, and the old adage that "quality cannot be tested in" is still very relevant. A tester cannot show up at the last minute and make a bad product good; high quality has to be part of the process from the beginning.

This section discusses how to integrate testing with development, where testing activities begin as soon as development activities begin, and are carried out in parallel with the development stages. Specific activities, including planning, active testing, and development-influencing activities, can be associated with each of the traditional lifecycle phases. These activities can be carried out by the developers or by separate test engineers, and can be associated with development stages

within the confines of any specific development process. These testing activities allow the tester to detect and prevent faults throughout the software development process.

Projects that begin test activities after implementation is complete often produce very unreliable software. Wise testers (and testing level 4 organizations) incorporate a chain of test plans and procedures that begin in the first steps of software development, and proceed through all subsequent steps. By integrating software testing activities into all stages of the software development lifecycle, we can make dramatic improvements in the effectiveness and efficiency of testing, and impact the software development process in such a way that high quality software is more likely to be built.

Other textbooks and the research literature contain dozens of software processes (waterfall, spiral, evolutionary-prototyping, eXtreme programming, etc.). This section uses the following distinct stages without assuming any order or mapping them onto a specific process. Thus, the suggestions in this section can be adapted to whatever process is being used.

1. Requirements analysis and specification
2. System and software design
3. Intermediate design
4. Detailed design
5. Implementation
6. Integration
7. System deployment
8. Operation and maintenance

Any development process involves communication, comprehension, and transition of information among stages. Mistakes can be made during any stage, in the information handling, or in the transfer of the information from one stage to another. Integrating testing with the process is about trying to find errors at each stage as well as preventing these errors from propagating to other stages. Also, the integration of testing throughout the lifecycle provides a way to verify and trace consistencies among the stages. Testing should not be isolated into separate stages, but rather be on a parallel track that affects all stages.

Testing has different objectives during each stage, and these objectives are achieved in different ways. These sub-objectives of testing at each stage will then achieve the overall objective of ensuring high quality software. For most stages, the testing activities can be broken into three broad categories: *test actions* – testing the product or artifacts created at that stage; *test design* – using the development artifacts of that stage or testing artifacts from the previous stage to prepare to test the final software; and *test influence* – using development or test artifacts to influence future development stages.

6.3.1 Requirements Analysis and Specification

A software requirements and specifications document contains a complete description of the external behavior of the software system. It provides a way to

Table 6.1. Testing objectives and activities during requirements analysis and specification

Objectives	Activities
Ensure requirements are testable Ensure requirements are correct Ensure requirements are complete Influence the software architecture	Set up testing requirements ■ testing criteria ■ support software needed ■ testing plans at each level ■ build test prototypes Clarify requirement items and test criteria Develop project test plan

communicate with the other stages of the development, and defines the contents and boundary of the software system. Table 6.1 summarizes the major objectives and activities during requirements analysis and specification.

The major **test action goal** is to evaluate the requirements themselves. Each requirement should be evaluated to ensure it is correct, testable, and that the requirements together are complete. Many methods have been presented to do this, most commonly inspections and prototyping. These topics are well described elsewhere and are explicitly not covered in this book. A key point is that the requirements should be evaluated *before* design starts.

The major **test design goal** is to prepare for system testing and verification activities. Test requirements should be written to state testing criteria for the software system and high-level test plans should be developed to outline the testing strategy. The test plan should also include the scope and objectives for testing at each stage. This high-level test plan will be referenced in the later detailed test plans. The testing requirements should describe support software needed for testing at each stage. Testing requirements must be satisfied by later testing.

The major **test influence goal** is to influence the software architectural design. Project test plans and representative system test scenarios should be built to show that the system meets the requirements. The process of developing the test scenarios will often help detect ambiguous and inconsistent requirements specifications. The test scenarios will also provide feedback to the software architectural designers and help them develop a design that is easily testable.

6.3.2 System and Software Design

System and software design partitions the requirements into hardware or software systems and builds the overall system architecture. The software design should represent the software system functions so that they can be transformed into executable programs. Table 6.2 summarizes the major objectives and activities during system and software design.

The major **test action goal** is to verify the mapping between the requirements specification and the design. Any changes to the requirements specification should be reflected in the corresponding design changes. Testing at this stage should help validate the design and interface.

Table 6.2. Testing objectives and activities during system and software design

Objectives	Activities
Verify mapping between requirements specification and system design	Validate design and interface
Ensure traceability and testability	Design system tests
Influence interface design	Develop coverage criteria
	Design acceptance test plan
	Design usability test (if necessary)

The major **test design goal** is to prepare for acceptance and usability testing. An acceptance test plan is created that includes acceptance test requirements, test criteria, and a testing method. Also, requirements specifications and system design specifications should be kept traceable and testable for references and changes for the later stages. Testing at the system and software design stage also prepares for unit testing and integration testing by choosing coverage criteria from the previous chapters.

The major **test influence goal** is to influence the design of the user interface. Usability tests or an interface prototype should be designed to clarify the customer's interface desires. Usability testing is carried out when the user interface is an integral part of the system.

6.3.3 Intermediate Design

In intermediate design, the software system is broken into components and classes associated with each component. Design specifications are written for each component and class. Many problems in large software systems arise from component interface mismatches. The major **test action goal** is to avoid mismatches of interfaces. Table 6.3 summarizes the major objectives and activities during intermediate design.

The major **test design goal** is to prepare for unit testing, integration testing, and system testing by writing the test plans. The unit and integration test plans are refined at this level with information about interfaces and design decisions. To prepare for testing at the later stages, test support tools such as test drivers, stubs, and testing measurement tools should be acquired or built.

The major **test influence goal** is to influence detailed design. The class integration and test order (CITO) from Section 6.2.2 should be determined so as to have the proper effect on the detailed design.

Table 6.3. Testing objectives and activities during intermediate design

Objectives	Activities
Avoid mismatches of interfaces	Specify system test cases
Prepare for unit testing	Develop integration and unit test plans
	Build or collect test support tools
	Suggest ordering of class integration

Table 6.4. Testing objectives and activities during detailed design

Objectives	Activities
Be ready to test when modules are ready	Create test cases (if unit) Build test specifications (if integration)

6.3.4 Detailed Design

At the detailed design stage, testers write subsystem specifications and pseudocode for modules. Table 6.4 summarizes the major objectives and activities during detailed design. The major **test action goal** at the detailed design stage is to make sure that all test materials are ready for testing when the modules are written. Testers should prepare for both unit and integration testing. Testers must refine detailed test plans, generate test cases for unit testing, and write detailed test specifications for integration testing. The major **test influence goal** is to influence the implementation and unit and integration testing.

6.3.5 Implementation

At some point during software development, the "rubber hits the road" and the programmers start writing and compiling classes and methods. Table 6.5 summarizes the major objectives and activities during implementation.

The major **test action goal** is to perform effective and efficient unit testing. The effectiveness of unit testing is largely based on the testing criterion used and test data generated. Unit testing performed at this stage is as specified by the unit test plan, testing criteria, test cases, and test support tools that were made ready at the earlier stages. Unit test results and problems should be saved and reported properly for further processing. Designers and developers whose duties are becoming lighter at this point should be made available to help testers.

The major **test design goal** is to prepare for integration and system testing. The major **test influence goal** is that efficient unit testing can help ensure early integration and system testing. It is much cheaper and easier to find and fix bugs during unit testing!

Table 6.5. Testing objectives and activities during implementation

Objectives	Activities
Efficient unit testing Automatic test data generation	Create test case values Conduct unit testing Report problems properly

Table 6.6. Testing objectives and activities during integration

Objectives	Activities
Efficient integration testing	Perform integration testing

6.3.6 Integration

The CITO problem, discussed in the previous section, is the major issue during software integration. Table 6.6 summarizes the major objectives and activities during integration.

The major **test action goal** is to perform integration testing. Integration and integration testing begin as soon as the needed components of an integrated subsystem pass unit testing. In practice, the CITO problem can be solved in a pragmatic way by integrating classes as soon as they are delivered from unit testing. Integration testing is concerned with finding errors that result from unexpected interactions among components.

6.3.7 System Deployment

Table 6.7 summarizes the major objectives and activities during system deployment. The major **test action goal** is to perform system testing, acceptance testing, and usability testing. System testing has the particular purpose to compare the software system to its original objectives, in particular, validating whether the software meets the functional and non-functional requirements. System testing test cases are developed from the system and project test plan from the requirements specification and software design phase according to criteria covered in previous chapters. Acceptance testing can be started as soon as system testing is completed. Acceptance testing ensures that the complete system satisfies the customers' needs, and should be done with their involvement. Test cases are derived from acceptance test plans and test data set up previously. Usability testing evaluates the user interface of the software. It should also be done with user involvement.

6.3.8 Operation and Maintenance

After the software is released (or delivered, or deployed, or whatever), users will occasionally find new problems or request new features. When the software is changed, it must be regression tested. Regression testing helps ensure that the

Table 6.7. Testing objectives and activities during system deployment

Objectives	Activities
Efficient system testing	Perform system testing
Efficient acceptance testing	Perform acceptance testing
Efficient usability testing	Perform usability testing

Table 6.8. Testing objectives and activities during operation and maintenance

Objectives	Activities
Efficient regression testing	Capture user problems Perform regression testing

updated software still possesses the functionality it had before the updates, as well as the new or modified functionality. Table 6.8 summarizes the major objectives and activities during operation and maintenance.

6.3.9 Summary

A key factor to instilling quality into a development process is based on individual professional ethics. Developers and testers alike can choose to **put quality first**. If the process is such that the tester does not know how to test it, then don't build it. This will sometimes result in conflicts with time-driven management, but even if you lose the argument, you will gain respect. It is important that developers begin test activities early. It also helps to take a stand against taking shortcuts. Almost all projects will eventually be faced with taking shortcuts that will ultimately reduce the quality of the software. Fight it! If you lose the argument you will gain respect: document your objections, vote with your feet, and don't be afraid to be right!

It is also essential that test artifacts be managed. A lack of organization is a sure recipe for failure. Put test artifacts under version control, make them easily available, and update them regularly. These artifacts include test design documents, tests, test results, and automated support. It is important to keep track of the criteria-based source of the tests, so when the source changes, it is possible to track which tests need to change.

6.4 TEST PLANS

A major emphasis for many organizations is documentation, including test plans and test plan reporting. Unfortunately, putting too much of a focus on documentation can lead to an environment where lots of meaningless reports are produced but nothing useful is done. That is why this book focuses on content, not form. The contents of a test plan are essentially how the tests were created, why the tests were created, and how they will be run.

Producing test plans, however, is an essential requirement for many organizations. Companies and customers often impose templates or outlines. Rather than surveying many different types of test plans, we look at the IEEE standard definition. Unfortunately, this is quite old (1983!), but it is still the most widely known. A quick search on the Web will supply you with more test plans and test plan outlines than you could ever use. ANSI/IEEE Standard 829-1983 describes a test plan as:

"A document describing the scope, approach, resources, and schedule of intended testing activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning."

Several different general types of test plans are:

1. A *mission plan* tells "why." Usually only one mission plan appears per organization or group. Mission plans describe the reason that the organization exists, are typically very short (5–10 pages), and are the least detailed of all plans.
2. A *strategic plan* tells "what" and "when." Again, only one strategic plan usually is used per organization, although some organizations develop a strategic plan for each category of project. Strategic plans can say things such as "we will always do Edge Coverage during unit testing" and "Integration Testing will be driven by couplings." Strategic plans are more detailed and longer than mission plans, sometimes 20–50 pages or more. They are seldom detailed enough to be directly useful to practicing testers or developers.
3. A *tactical plan* tells "how" and "who." Most organizations use one overall tactical plan per product. Tactical plans are the most detailed of all plans, and are usually living documents. That is, a tactical plan may begin life as a table of contents, and be continually added to during the life of the product or product development. For example, a tactical test plan would specify how each individual unit will be tested.

Below are outlines of two sample test plans, provided as example only. The plans were derived from numerous samples that have been posted on the Web, so do not exactly represent a single organization. The first is for system testing and the second is a tactical plan for unit testing. Both are based on the IEEE 829-1983 standard.

1. Purpose
   The purpose of a **test plan** is to define the strategies, scope of testing, philosophy, test exit and entrance criteria, and test tools that will be used. The plan should also include management information such as resource allocations, staff assignments, and schedules.
2. Target Audience and Application
   (a) The test staff and quality assurance personnel must be able to understand and implement the test plan.
   (b) The quality assurance personnel must be able to analyze the results and make recommendations on the quality of the software under test to management.
   (c) The developers must be able to understand what functionalities will be tested and the conditions under which the tests are to be performed.
   (d) The marketing personnel must be able to understand with which configurations (hardware and software) the product was tested.
   (e) Managers must understand the schedule to the degree of when testing is to be performed and when it will be finished.
3. Deliverables
   The results of testing are the following deliverables:
   (a) Test cases, including input values and expected results
   (b) Test criteria satisfied
   (c) Problem reports (generated as a result of testing)
   (d) Test coverage analysis

4. Information Included
   Each test plan should contain the following information. Note that this can (and often does) serve as the outline of the actual test plan, and can be tailored to most environments successfully.
   (a) Introduction
   (b) Test items
   (c) Features tested
   (d) Features not tested (per cycle)
   (e) Test criteria, strategy and approach
      - Syntax
      - Description of functionality
      - Argument values for tests
      - Expected output
      - Specific exclusions
      - Dependencies
      - Test case success criteria
   (f) Pass/fail standards
   (g) Criteria for beginning testing
   (h) Criteria for suspending test and requirements for restarting
   (i) Test deliverables/status communications documents
   (j) Hardware and software requirements
   (k) Responsibilities for determining problem severity and correcting problems
   (l) Staffing and training needs
   (m) Test schedules
   (n) Risks and contingencies
   (o) Approvals

The above plan is in a very general and high level style. The next example is in a much more detailed style, and is more suited for tactical test plans for engineers.

1. Purpose
   The purpose of the test plan is to describe the scope, approach, resources, and schedule of all testing activities. The plan should identify the items to be tested, the features to be tested, the testing tasks to be performed, the personnel responsible for each task, and the risks associated with this plan.

   The test plan should be a dynamic document that can be used by testers, managers, and developers. The test plan should evolve as the project evolves. At the end of the project the test plan should document the activities and be the vehicle by which all parties sign indicating approval of the final product.
2. Outline
   A test plan has the following structure:
   (a) Test-plan identifier
   (b) Introduction
   (c) Test reference items
   (d) Features that will be tested

(e) Features that will not be tested
(f) Approach to testing
(g) Criteria for pass/fail
(h) Criteria for suspending test and requirements for restarting
(i) Test deliverables
(j) Testing tasks
(k) Environmental needs
(l) Responsibilities
(m) Staffing and training needs
(n) Schedule
(o) Risks and contingencies
(p) Approvals

The sections are ordered in the sequence above. Additional sections may be included if necessary. If some or all of the content of a section is in another document, a reference to that document can be listed. The referenced material must be easily available. The following sections give details on the content of each section.

3. Test-Plan Identifier
   Give a unique identifier (name) to this test plan.
4. Introduction
   Give a description or purpose of the software, so that both the tester and the client are clear as to the purpose of the software and the approach to be taken in testing.
5. Test Reference Items
   Identify items that are referred to by the tests, including their version/revision and dates. Supply references to the following documents, if available:
   (a) Requirements specification
   (b) Design specification
   (c) Users guide
   (d) Operations guide
   (e) Installation guide
   (f) Analysis diagrams, including data flow, etc.
   (g) UML or other modeling documents
6. Features that Will Be Tested
   Identify all features and feature combinations that need to be tested. Identify the test design that is associated with each feature and each combination of features.
7. Features that Will not Be Tested
   Identify all features and significant combinations of features that will not be tested. Most importantly, state why.
8. Approach to Testing
   For each major group of features or feature combinations, specify the approach that will ensure that these feature groups are adequately tested. Specify the major activities, criteria, and tools that will be used.

   The approach should be described in enough detail to identify the major testing tasks and estimate how long each will take.

9. Criteria for Pass/Fail
   Specify the measure to be used to determine whether each test item has passed or failed testing. Will it be based on a criterion? The number of known faults?
10. Criteria for Suspending Testing and Requirements for Restarting
    In certain situations, testing must stop and the software sent back to the developers. Specify the criteria used to suspend all or any portion of the testing. Specify the activities that must be repeated to resume or restart testing activities.
11. Test Deliverables
    Identify the documents that should be included in the report. The following are candidate documents.
    (a) Test plan
    (b) Test design specifications
    (c) Test case specifications
    (d) Test process
    (e) Test logs
    (f) Test trouble reports
    (g) Test summary reports
    (h) Test input data and test output data (or where they are located)
12. Testing Tasks
    Identify the tasks necessary to prepare for and perform testing. Identify all dependencies among the tasks.
13. Environmental Needs
    Specify both the necessary and desired properties of the test environment. This specification should contain:
    (a) The physical characteristics of the facilities, including the hardware
    (b) Any communications and system software
    (c) The mode of usage (stand-alone, transient, web-based, etc.)
    (d) Any other software or supplies needed to run the test
    (e) Test tools needed
    (f) Any other testing needs (e.g., publications) and where to get them
14. Responsibilities
    Identify the groups responsible for all aspects of testing and correcting problems. In addition, identify the groups responsible for providing the test reference items identified and the environmental needs above.
15. Staffing and Training Needs
    Specify test staffing needs in terms of knowledge and skill. Identify training options when appropriate and necessary.
16. Schedule
    Include all test milestones identified in the software project schedule. Define any additional test milestones needed. Estimate the time required to do each testing task and specify the schedule for each testing task and test milestone.
17. Risks and Contingencies
    Identify any risky assumptions of the test plan. For example, specialized

knowledge may be needed but not available. Specify contingency plans for each.

18. Approvals
    Specify the names and titles of all persons who must approve this plan, and include space for them to sign and date the document.

6.5 IDENTIFYING CORRECT OUTPUTS

The main contribution of this book is set of coverage criteria for testing. But no matter what coverage criterion is used, sooner or later one wants to know whether a given program executes correctly on a given input. This is the *oracle* problem in software testing.

The oracle problem can be surprisingly difficult to solve, and so it helps to have a range of approaches available. This section describes several common approaches to the oracle problem.

6.5.1 Direct Verification of Outputs

If you are lucky, your program will come with a specification, and the specification will be clear as to what output accompanies a given input. For example, a *sort* program should produce a permutation of its input in a specified order.

Having a human evaluate the correctness of a given output is often effective, but is also expensive. Naturally, it is cheaper to automate this process. Automating direct verification of the output, when possible, is one of the best methods of checking program behavior. Below is an outline of a checker for sorting. Notice that the checking algorithm is *not* another sorting algorithm. It is not only different, it is not particularly simple. That is, writing output checkers can be hard.

```
Input: Structure S
    Make copy T of S
    Sort S
     // Verify S is a permutation of T
     Check S and T are of the same size
     For each object in S
          Check if object appears in S and T same number of times
     // Verify S is ordered
     For each index i but last in S
          Check if S[i] <= S[i+1]
```

Unfortunately, direct verification is not always possible. Consider a program that analyzes Petri nets, which are useful for modeling processes with state. One output of such analysis is the probability of being in any given state. It is difficult to look at a given probability and assert that it is correct – after all, it is just a number. How do you know if all of the digits are, in fact, the right ones? For Petri nets, the final probabilities cannot easily be related back to the input Petri net.

6.5.2 Redundant Computations

When direct verification is not applicable, redundant computations can be used instead. For example, to evaluate automatically the correctness of a *min* program, one could use another implementation of *min*–preferably a trustworthy or "gold" version. This initially appears to be a circularity; why should one trust one implementation more than another?

Let us formalize the process. Suppose that the program under test is labeled P, and $P(t)$ is the output of P on test t. A specification S of P also specifies an output $S(t)$, and we usually demand that $S(t) = P(t)$.[1] Suppose that S is, itself, executable, thereby allowing us to automate the output checking process. If S itself contains one or more faults, a common occurrence, $S(t)$ may very well be incorrect. If $P(t)$ is incorrect in exactly the same way, the failure of P goes undetected. If P fails in some way that is different from S on some test t, then the discrepancy will be investigated, with at least the possibility that the faults in both S and P will be discovered.

A potential problem is when P and S have faults that result in incorrect and identical (and hence unremarkable) outputs. Some authors have suggested that the oracle S should be developed independently of P to reduce this possibility. From a practical standpoint, such independent development is difficult to achieve.

Further, independent development is very unlikely to lead to independent failures. Both experimental evidence and theoretical arguments suggest that common failures occur at a rate substantially above what would be expected given an assumption of independence. The basic reason for this is that some inputs are "harder" than others to get right, and it is precisely these inputs that are the most likely to trigger common failures across multiple implementations.

Still, testing one implementation against another is an effective, practical technique for testing. In industry, the technique is implemented most often in regression testing, where the executable version of a specification S is simply the previous release of the software. Regression testing is extremely effective at identifying problems in software, and should be a standard part of any serious commercial software development activity.

Sometimes a problem might have different algorithms to solve it, and implementations of the different algorithms are excellent candidates for checking against each other, even though the common failure problem still remains. For example, consider searching algorithms. A binary search routine could easily be tested by comparing the result with a linear search.

6.5.3 Consistency Checks

An alternative to direct verification or redundant computations is consistency analysis. Consistency analysis is typically incomplete. Consider the Petri net example again. Given a putative probability, one can certainly say that if it is negative or larger than unity, then it is wrong. Consistency analysis can also be internal. Recall the RIP (reachability, infection, propagation) model for failures from Chapter 1. External checks can only examine the outputs, so the infection must propagate for the error to be detected.

Internal checks raise the possibility of identifying faulty behavior with only the first two (RI) properties. It is quite common for programmers to require certain relations to hold on internal structures. For example, an object representation might require that a given container never hold duplicate objects. Checking these "invariant" relations is an extremely effective way of finding faults. Programmers trained in developing software under the contract model can produce the code for such checking in the course of normal development. For object-oriented software, such checks are typically organized around object invariants – both on the abstraction of the object and on its representation – as well as object method preconditions and postconditions. Tools such as assertion facilities can efficiently turn such checks on during testing and turn them back off, if necessary for performance, during operation.

6.5.4 Data Redundancy

An extremely powerful method of evaluating correctness on a given input is to consider how the program behaves on other inputs. Consider a computation of the *sine* function. Given a computation $sin(x)$ for some input x, it is quite difficult to decide if the output is exactly right. Comparing to another implementation of *sine* is one possibility, but other techniques are possible. If *sine* is available, it is likely that *cosine* is as well, and it is an identity that $sin(x)^2 + cos(x)^2 = 1$, for all values of x.

This last check helps in many cases, but it doesn't help in the case where both $sin(x)$ and $cos(x)$ happen to be wrong in compensating ways. For example, if *cos* happens to be implemented with a call to *sin*, then we have not made much progress.

Still, identities are an extremely useful approach. Further, they often work with classes. For example, adding an element to a container and then removing the element from the container often has a well-defined effect on the container. For some containers, such as bags, the result is no change at all. For other containers, such as sets, the result might be no change, or it might be a change of one element, depending on whether the item was originally in the container.

Even more powerful are identities that use the same program, but on different inputs. Consider $sin(x)$ again. Another identity is $sin(a + b) = sin(a)cos(b) + cos(a)sin(b)$.[2] We have a relation on the inputs (namely, $a + b = x$) and a relation on the outputs ($sin(x)$ is a simple expression in terms of *sine* applied to a and b). Such checks can be repeated as often as desired with different random choices for the value of a. It turns out that even the most malicious implementor of a *sine* function has a vanishingly small chance of fooling such a check. This is truly powerful output checking!

This method applies, in its most powerful form, only to well-behaved mathematical functions such as *sine*. However, the approach applies to many other types of software. Consider an implementation of TCAS, the Traffic Collision and Avoidance System deployed on commercial aircraft. The function of this system is to give pilots guidance as to how best to avoid a potential collision. In the "vertical resolution" mode, the outputs of TCAS, or "resolution advisories," are either to stay level, to climb, or to descend.

TCAS is a complex system that considers many factors, including multiple recent positions of the various aircraft, the existence of complementary TCAS processing on other aircraft, proximity to the ground, and so on.

To apply the technique of data redundancy, suppose we rerun the TCAS software with slightly different positions for some or all of the aircraft. We would expect, in many cases, that the resolution advisory would not change. If the resolution advisory appeared to be unstable for some closely related inputs, we would have a strong indication that the pilot might not wish to place much confidence in the resolution advisory. Back in the laboratory, the TCAS engineers might want to pay special attention to such inputs – perhaps even regarding them as failures.

The technique illustrated with the TCAS software can be applied to any software where the input space has some notion of continuity. In any such system, it makes sense to speak of "nearby" inputs, and in many such systems, the output can be expected to vary in some piecewise continuous way.

6.6 BIBLIOGRAPHIC NOTES

Binder [33] has an excellent and detailed practical description of regression testing, in which he claimed that unautomated regression testing is equivalent to no regression testing. Rothermel and Harrold published the regression testing framework of inclusiveness, precision, efficiency, and generality [303], and evaluated a safe technique empirically [153]. Later papers by Li, Harman, and Hierons [208] and Xie and Notkin [361] are good places to start into the regression testing literature.

The notions of stubs and drivers have been around for decades. They were discussed in books as far back as the 1970s [88, 104, 165, 249]. Beizer pointed out that the creation of stubs can be error prone and costly [29]. Current tools such as JUnit use the term "mock" as a specialized form of a stub [27].

The first paper that defined the CITO problem was by Kung et al. [197]. They showed that when classes do not have any dependency cycles, deriving an integration order is equivalent to performing a topological sorting of classes based on their dependency graph – a well-known graph theory problem. In the presence of dependency cycles, they proposed a strategy of identifying strongly connected components (SCCs) and removing associations until no cycles remain. When there is more than one candidate for cycle breaking, Kung et al.'s approach chooses randomly.

Most researchers [197, 48, 321, 329] estimated the cost CITO by counting the number of test stubs that need to be created during integration testing. This method assumes that all stubs are equally difficult to write. Briand et al. pointed out that the cost of stubs is not constant and developed a genetic algorithm to solve the CITO problem [44]. Malloy et al. first tried to consider test stub complexity when estimating the testing effort [223].

Briand et al. showed that the complexity of stub construction for parent classes is induced by the likely construction of stubs for most of the inherited member functions [47]. Abdurazik and Offutt developed a new algorithm, based on coupling analysis, that uses more information about how the stubbed class couples with other classes to find a cheaper ordering [3].

Some good sources for details about test process and accepted definitions of terms are IEEE standards [175], BCS standards [317], books by Hetzel [160], DeMillo et al. [100], Kaner, Falk, and Nguyen [182], Dustin, Rashka, and Paul [109], and Copeland [90].

Weyuker [341] wrote an early essay identifying the oracle problem and various approaches to addressing it. Both Meyer [241] and Liskov [355] talk about how to articulate checkable assertions in the context of the contract model. Several commercial tools support assertion checking.

The notion of building multiple versions was championed in the fault-tolerance context by a number of authors, most vocally by Avizienis [22]. Limits on reliability for multiversion software were first explored experimentally by Knight and Leveson [188], then theoretically by Eckhardt and Lee [110] and Littlewood and Miller [214], and in a different context by Geist et al. [133]. Multiversion software actually works better for testing than for fault tolerance. If two versions of the program behave differently on the same inputs, then we know we have found a good test, and at least one of the versions is wrong. In particular, it is helpful to view regression testing as a multiversion testing arrangement.

Blum and Kannan [38] and Lipton [210] give theoretical treatments of data redundancy for certain mathematically well-defined problems; Ammann and Knight [18] provide a less powerful, but more widely applicable, approach.

NOTES

1 If S is *underdetermined*, then the requirement $S(t) = P(t)$ is not correct. Instead, S should be viewed as allowing a set of possible outputs, and the correctness constraint is that P produces one of them, namely $P(t) \in S(t)$.

2 If we wish, we can rewrite the $cos(x)$ calls to $sin(\pi/2 - x)$ calls, but this is not strictly necessary.

7

Engineering Criteria for Technologies

This chapter discusses how to engineer the criteria from Chapters 2 through 5 to be used with several different types of technologies. These technologies have come to prominence after much of the research literature in software testing, but are now very common and account for a large percentage of new applications being built. Sometimes we modify the criteria, and sometimes simply discuss how to build the models that the existing criteria can be applied to. Some of these technologies, such as Web applications and embedded software, tend to have extremely high reliability requirements. So testing is crucial to the success of the applications. The chapter explains what is different about these technologies from a testing viewpoint, and summarizes some of the existing approaches to testing software that uses the technologies.

Object-oriented technologies became prominent in the mid-1990s and researchers have spent quite a bit of time studying their unique problems. A number of issues with object-oriented software have been discussed in previous chapters, including various aspects of applying graph criteria in Chapter 2, integration mutation in Chapter 5 and the CITO problem in Chapter 6. This chapter looks into how the use of classes affects testing, and focuses on some challenges that researchers have only started addressing. Most of these solutions that have not yet made their way into automated tools. The most important of these challenges is testing for problems in the use of inheritance, polymorphism, and dynamic binding.

One of the most active areas in terms of technology as well as testing research is that of web applications and web services. Most web software is object-oriented in nature, but the Web allows some very interesting[1] structures that require testers to adapt their techniques and criteria. One interesting aspect of web applications and services is that they have to work **very well** – the environment is quite competitive and users will not tolerate failures. Web applications also have stringent security requirements, which is discussed in Chapter 9. Web applications are built with a particular type of graphical user interface, HTML running in a browser, but testing general GUIs brings in additional complexities. Some of the ideas in this chapter are still in the research and development stage, so may not be ready for practical use.

Finally, this chapter discusses some of the issues with testing real-time and embedded software. They are combined because many systems incorporate both. The amount of embedded software is growing very quickly as it pops up in all sorts of mechanical and electronic devices. Many of these systems also have safety critical requirements, a topic we defer until Chapter 9.

7.1 TESTING OBJECT-ORIENTED SOFTWARE

Object-oriented languages emphasize defining abstractions in the software. Abstractions such as abstract data types model concepts in an application domain. These abstractions are implemented in classes that represent user-defined types that have both state and behavior. This approach to abstraction has many benefits, but it also changes how testing needs to be carried out. The most important factor is that object-oriented software shifts much of the **complexity** of our software from algorithms in units and methods to how we connect software components. Thus, we need less emphasis on unit testing and more on integration testing.

Another factor is that the relationships in object-oriented components tend to be very complex. The compositional relationships of inheritance and aggregation, especially when combined with polymorphism, introduce new kinds of faults and require new methods for testing. This is because the way classes and components are integrated is different in object-oriented languages.

Object-oriented languages use *classes* (data abstraction), *inheritance*, *polymorphism*, and *dynamic binding* to support abstraction. New types created by inheritance are *descendants* of the existing type. A class *extends* its parent class if it introduces a new method name and does not override any methods in an ancestor class. (This new method is called an *extension* method.) A class *refines* the parent class if it provides new behavior not present in the overridden method, does not call the overridden method, and its behavior is semantically consistent with that of the overridden method.

Programmers use two types of inheritance, subtype and subclass. If class **B** uses *subtype* inheritance from class **A**, then it is possible to freely substitute any instance of class **B** for an instance of **A** and still satisfy an arbitrary client of class **A**. This is called the *substitution principle*. In other words, **B** has an "is-a" relationship with **A**. For example, a *chair* "is a" special case of a *furniture*. *Subclass* inheritance allows descendant classes to reuse methods and variables from ancestor classes without necessarily ensuring that instances of the descendants meet the specifications of the ancestor type. Although there has been intense discussion of which use of inheritance is appropriate, from a tester's perspective, professional programmers use both types. In the case of subtype inheritance, testers should focus on verifying that the substitution principle holds. The lack of firm guiding principles in subclass inheritance provides testers ample opportunities to find faults.

These abstractions have major effects on component integration. If class **B** inherits from class **A**, and both **A** and **B** define a method **m()**, then **m()** is called a *polymorphic method*. If an object **x** is *declared* to be of type **A** (in Java, "**A x;**"), then during execution **x** can have either the *actual* type **A** (from "**x = new A();**") or **B** (from "**x = new B();**"). When a call is made to a polymorphic method (for example, "**x.m();**"), which version is executed depends on the **current** actual type of

the object. The collection of methods that can be executed is called the *polymorphic call set* (PCS). In this example, the PCS for **x.m()** is {**A::m()**, **B::m()**}.

7.1.1 Unique Issues with Testing OO Software

Several testing issues are unique to object-oriented software. Some researchers have claimed that traditional testing techniques are not as effective for object-oriented software and sometimes test the wrong things. Methods tend to be smaller and less complex, so path-based testing techniques may be less useful. As discussed previously, inheritance, polymorphism, and dynamic binding introduce special problems. The execution path is no longer based on the class's static declared type, but the dynamic type; and that is not known until execution.

When testing object-oriented software, a class is usually regarded as the basic unit of testing. This leads to four levels of testing classes.

1. *Intra-method testing*: Tests are constructed for individual methods (this is traditional unit testing)
2. *Inter-method testing*: Multiple methods within a class are tested in concert (this is traditional module testing)
3. *Intra-class testing*: Tests are constructed for a single class, usually as sequences of calls to methods within the class
4. *Inter-class testing*: More than one class is tested at the same time, usually to see how they interact (this is a type of integration testing)

Early research in object-oriented testing focused on the inter-method and intra-class levels. Later research focused on the testing of interactions between single classes and their users and system-level testing of OO software. Problems associated with inheritance, dynamic binding and polymorphism cannot be addressed at the inter-method or intra-class levels. These require multiple classes that are coupled through inheritance and polymorphism, that is, inter-class testing.

Most research in object-oriented testing has focused on one of two problems. One is the ordering in which classes should be integrated and tested. The CITO problem was discussed in Chapter 6. The other is developing techniques and coverage criteria for selecting tests. These coverage criteria are refinements of one or more of the criteria presented in the earlier chapters.

7.1.2 Types of Object-Oriented Faults

One of the hardest tasks for object-oriented software engineers is **visualizing** the interactions that can occur in the presence of inheritance, polymorphism, and dynamic binding. They are often very complex! This visualization assumes a class encapsulates state information in a collection of *state variables*, and has a set of behaviors that are implemented by methods that use those state variables.

As an example, consider the UML class diagram and code fragment shown in Figure 7.1. In the figure, V and X extend W, V overrides method m(), and X overrides methods m() and n(). The minuses ("–") indicate the attributes are private and the pluses ("+") indicate the attributes are non-private. The declared type of *o* is *W*, but at line 10, the actual type can be either *V* or *W*. Since *V*

Figure 7.1. Example class hierarchy in UML.

overrides *m*(), which version of *m*() is executed depends on the input flag to the method *f*().

To illustrate problems from method overriding and polymorphism, consider the simple inheritance hierarchy shown on the left of Figure 7.2. The root class **A** contains four state variables and six methods. The state variables are *protected*, which means they are available to **A**'s descendents (**B** and **C**). **B** declares one state variable and three methods and **C** declares three methods. The arrows on the figure show the overriding: **B::h()** overrides **A::h()**, **B::i()** overrides **A::i()**, **C::i()** overrides **B::i()**, **C::j()** overrides **A::j()**, and **C::l()** overrides **A::l()**. The table on the right of Figure 7.2 shows the state variable definitions and uses for some of the methods in the hierarchy. The problem begins with a call to **A::d()**. This small example has

Method	Defs	Uses
A::h	{A::u, A::w}	
A::i		{A::u}
A::j	{A::v}	{A::w}
A::l		{A::v}
B::h	{B::x}	
B::i		{B::x}
C::i	{C::y}	
C::j		{C::y}
C::l		{A::v}

Figure 7.2. Data flow anomalies with polymorphism.

Figure 7.3. Calls to d() when object has various actual types.

some very complex interactions that potentially yield some very difficult problems to model, understand, test, and debug.

Suppose that an instance of **A** is bound to an object **o** (**o = new A();**), and a call is made through **o** to **A::d()** (**o.d()**), which calls **A::g()**, which calls **A::h()**, which calls **A::i()**, which finally calls **A::j()**. In this case, the variables **A::u** and **A::w** are first defined, then used in **A::i()** and **A::j()**, which poses no problems.

Now suppose that an instance of **B** is bound to **o** (**o = new B();**), and a call to **d()** is made (**o.d()**). This time **B**'s version of **h()** and **i()** are called, **A::u** and **A::w** are **not** given values, and thus the call to **A::j()** can result in a data flow anomaly!

Visualizing Polymorphism with the Yo-Yo Graph

Understanding which version of a method **will** be executed and which versions **can** be executed is very difficult for developers and testers alike. Execution can "bounce" up and down among levels of inheritance, which is called the yo-yo effect. The yo-yo *graph* is defined on an inheritance hierarchy that has a root and descendents. The graph shows all new, inherited, and overridden methods for each descendent. Method *calls* in the source are represented as arrows from caller to callee. Each class is given a *level* in the yo-yo graph that shows the actual calls made if an object has the actual type of that level. Bold arrows are actual calls and light arrows are calls that cannot be made due to overriding.

Consider the inheritance hierarchy from Figure 7.2. Assume that in **A**'s implementation, **d()** calls **g()**, **g()** calls **h()**, **h()** calls **i()**, and **i()** calls **j()**. Further, assume that in **B**'s implementation, **h()** calls **i()**, **i()** calls its parent's (that is, **A**'s) version of **i()**, and **k()** calls **l()**. Finally, assume that in **C**'s implementation, **i()** calls its parent's (this time **B**'s) version of **i()**, and **j()** calls **k()**.

Figure 7.3 is a yo-yo graph of this situation and illustrates the **actual** sequence of calls if a call is made to **d()** through an instance of actual type **A**, **B**, and **C**. The top level of the graph assumes that a call is made to method **d()** through an object of actual type **A**. This sequence of calls is simple and straightforward. The second level shows the situation is more complex when the object is of actual type **B**. When

Table 7.1. Faults and anomalies due to inheritance and polymorphism

Acronym	Fault/Anomaly
ITU	Inconsistent type use (context swapping)
SDA	State definition anomaly (possible post-condition violation)
SDIH	State definition inconsistency (due to state variable hiding)
SDI	State defined incorrectly (possible post-condition violation)
IISD	Indirect inconsistent state definition
ACB1	Anomalous construction behavior (1)
ACB2	Anomalous construction behavior (2)
IC	Incomplete construction
SVA	State visibility anomaly

g() calls **h()**, the version of **h()** defined in **B** is executed (the light dashed line from **A::g()** to **A::h()** emphasizes that **A::h()** is **not** executed). Then control continues to **B::i()**, **A::i()**, and then to **A::j()**.

When the object is of actual type **C**, we can see where the term "yo-yo" comes from. Control proceeds from **A::g()** to **B::h()** to **C::i()**, then back up through **B::i()** to **A::i()**, back to **C::j()**, back up to **B::k()**, and finally down to **C::l()**.

This example illustrates some of the complexities that can result in object-oriented programs due to method overriding and polymorphism. Along with this induced complexity comes more difficulty and effort required in testing.

Categories of Inheritance Faults and Anomalies

Inheritance helps developers be more creative, be more efficient, and reuse previously existing software components. Unfortunately, it also allows a number of anomalies and potential faults that anecdotal evidence has shown to be some of the most difficult problems to detect, diagnose, and correct. Table 7.1 summarizes the fault types that result from inheritance and polymorphism. Most apply to all programming languages, although the language that is used will affect details of how the faults look.

As pointed out above, object-oriented faults are different from faults in non-OO software. The following discussion assumes an anomaly or fault is manifested through polymorphism in a context that uses an instance of the ancestor. Thus, we assume that instances of descendant classes can be substituted for instances of the ancestor.

In an inconsistent type use fault (ITU) a descendant class does not override any inherited method. Thus, there can be no polymorphic behavior. Every instance of a descendant class C that is used when an instance of the ancestor T is expected can only behave exactly like an instance of T. That is, only methods of T can be used. Any additional methods specified in C are hidden since the instance of C is being used as if it is an instance of T. However, anomalous behavior is still possible. If an instance of C is used in multiple contexts (that is, through coercion, say first as a T, then as a C, then a T again), anomalous behavior can occur if C has extension methods. In this case, one or more of the extension methods can call a method of T or directly define a state variable inherited from T. Anomalous

Method	CalledMethods
Vector::insertElementAt	
Vector::removeElementAt	
Stack::pop	Vector::removeElementAt
Stack::push	Vector::insertElementAt

State Variable Uses and Definitions	
Variable / Method	*Vector* / *array*
Vector::insertElementAt	*d, u*
Vector::removeElementAt	*d, u*
Stack::pop	*d*, u**
Stack::push	*d*, u**

Figure 7.4. ITU: Descendant with no overriding methods.

behavior will occur if either of these actions results in an inconsistent inherited state.

Figure 7.4 shows an example class hierarchy. Class *Vector* is a sequential data structure that supports direct access to its elements. Class *Stack* uses methods inherited from *Vector* to implement the stack. The top table summarizes the calls made by each method, and the bottom table summarizes the definitions and uses (represented as "d" and "u," respectively) of the state space of *Vector*.

The method *Stack::pop*() calls *Vector::removeElementAt*(), and *Stack::push*() calls *Vector::insertElementAt*(). These two classes clearly have different semantics. As long as an instance of *Stack* is used only as a *Stack*, there will not be any behavioral problems. Also, if the *Stack* instance is used only as a *Vector*, there will not be any behavioral problems. However, if the same object is sometimes used as a *Stack* and sometimes as a *Vector*, behavioral problems can occur.

The code fragment in Table 7.1.2 illustrates this problem. Three elements are pushed onto a *Stack s*, then the *Vector* method *g*() is called. Unfortunately, *g*() removes an element from the middle of the stack, which violates its semantics. Worse, the three pops after the call to *g*() no longer work. The fault is manifested when control reaches the first call to *Stack::pop*() at line 14. Here, the element removed from the stack is **not** the last element that was added, thus the stack integrity constraint will be violated. At the third call to *Stack::pop*(), the program will probably fail because the stack is empty.

In a **state definition anomaly fault** (**SDA**), the state interactions of a descendant are not consistent with those of its ancestor. The refining methods implemented in the descendant should leave the ancestor in a state that is equivalent to the state that the ancestor's overridden methods would have left the ancestor in. For this to

Table 7.2. ITU: Code example showing inconsistent type usage

```
public void f (Stack s)
{
String s1 = "s1";
String s2 = "s2";
String s3 = "s3";
...
s.push (s1);
s.push (s2);
s.push (s3);

g (s);

s.pop();
s.pop();
// Oops! The stack is empty!
s.pop();
...
}

public void g (Vector v)
{
// Remove the last element
v.removeElementAt (v.size()-1);
}
```

be true, the refining methods provided by the descendant must have the same state interactions as each public method that is overridden. From a data flow perspective, this means that the refining methods must provide definitions for the inherited state variables that are consistent with the definitions in the overridden method. If not, then a potential data flow anomaly exists. Whether or not an anomaly actually occurs depends upon the sequences of methods that are valid with respect to the ancestor.

Figure 7.5 shows an example class hierarchy and tables of definitions and uses. The parent is class W, and it has descendants X, and Y. W defines methods $m()$ and $n()$, each of which has the definitions and uses shown in the table. Assume that a valid method call sequence is $W::m()$ followed by $W::n()$. As the table of definitions and uses shows, $W::m()$ defines state variable $W::v$ and $W::n()$ uses it. Now consider the class X and its refining method $X::n()$. It also uses state variable $W::v$, which is consistent with the overridden method and with the method sequence given above. Thus far, there is no inconsistency in how X interacts with the state of W.

Now consider class Y and the method $Y::m()$, which overrides $W::n()$ through refinement. Observe that $Y::m()$ does not define $W::v$, as $W::m()$ does; but defines $Y::w$ instead. Now, a data flow anomaly exists with respect to the method sequence $m();n()$ for the state variable $W::v$. When this sequence of methods is called on an instance of Y, $Y::w$ is defined first (because $Y::m()$ executes), but then $W::v$ is used by method $X::n()$. Thus, the assumption made in the implementation of $X::n()$

	W::v	*W::u*	*X::x*	*Y::w*
W::l		*def*		
W::m	*def*			
W::n	*use*	*use*		
X::n	*use*		*def*	
Y::l		*def*		
Y::m				*def*

Figure 7.5. SDA, SDIH: State definition anomalies.

that *W::v* is defined by a call to *m*() prior to a call to *n*() no longer holds, and a data flow anomaly has occurred. In this particular example, a failure occurs since there is no prior definition of *W::v* when *Y* is the type of an instance being used. In general, the program might not fail at this point, but only create an incorrect state.

In a **state definition inconsistency due to state variable hiding fault** (**SDIH**), introducing a local state variable can cause a data flow anomaly. If a local variable *v* is introduced to a class definition and the name of the variable is the same as an inherited variable *v*, the inherited variable is hidden from the scope of the descendant (unless explicitly qualified, as in *super.v*). A reference to *v* will refer to the descendant's *v*. This is not a problem if all inherited methods are overridden since no other method would be able to implicitly reference the inherited *v*. However, this pattern of inheritance is the exception rather than the rule. Some methods are usually not overridden. A data flow anomaly can exist if a method that normally defines the inherited *v* is overridden in a descendant when an inherited state variable is hidden by a local definition.

As an example, again consider the class hierarchy shown in Figure 7.5. Suppose the definition of class *Y* has the local state variable *v* that hides the inherited variable *W::v*. Further suppose method *Y::m*() defines *v*, just as *W::m*() defines *W::v*. Given the method sequence *m*();*n*(), a data flow anomaly exists between *W* and *Y* with respect to *W::v*.

In a **state defined incorrectly fault** (**SDI**), an overriding method defines the same state variable *v* that the overridden method defines. If the computation performed by the overriding method is not semantically equivalent to the computation of the overridden method with respect to *v*, then subsequent state dependent behavior in

Figure 7.6. IISD: Example of indirect inconsistent state definition.

the ancestor can be affected, and the externally observed behavior of the descendant will be different from the ancestor. This is not a data flow anomaly, but it is a **potential** behavior anomaly.

In an **indirect inconsistent state definition fault** (**IISD**), a descendant adds an extension method that defines an inherited state variable. For example, consider the class hierarchy shown in Figure 7.6(a), where *Y* specifies a state variable *x* and method *m*(), and the descendant *D* specifies method *e*(). Since *e*() is an extension method, it cannot be directly called from an inherited method, (*T::m*()), because *e*() is not visible to the inherited method. However, if an inherited method is overridden, the overriding method (such as *D::m*() as depicted in Figure 7.6(b) can call *e*() and introduce a data flow anomaly by having an effect on the state of the ancestor that is not semantically equivalent to the overridden method (e.g., with respect to the variable *T::y* in the example). Whether an error occurs depends on which state variable is defined by *e*(), where *e*() executes in the sequence of calls made by a client, and what state-dependent behavior the ancestor has on the variable defined by *e*().

In an **anomalous construction behavior fault**, **version 1** (**ACB1**), the constructor of an ancestor class *C* calls a locally defined polymorphic method *f*(). Because *f*() is polymorphic, a descendant class *D* can override it. If *D* does, then *D*'s version of *f*() will execute when the constructor of *C* calls *f*(), not the version defined by *C*. To see this, consider the class hierarchy shown in the left half of Figure 7.7. Class C's constructor calls *C::f*(). Class *D* contains the overriding method *D::f*() that defines the local state variable *D::x*. There is no apparent interaction between *D* and *C* since *D::f*() does not interact with the state of *C*. However, *C* interacts with *D*'s state through the call that *C*'s constructor makes to *C::f*(). In most common object-oriented languages (including Java and C-Sharp), constructor calls to polymorphic methods execute the method that is closest to the instance that is being created. For the class *C* in the hierarchy in Figure 7.7, the closest version of *f*() to *C* is specified by *C* itself, and thus executes when an instance of *C* is being constructed. For *D*, the closest version is *D::f*(), which means that when an instance of *D* is being

Figure 7.7. ACB1: Example of anomalous construction behavior.

constructed, the call made to $f()$ in C's constructor actually executes $D::f()$ instead of its own locally specified $f()$. This is illustrated by the partial yo-yo graph in the right half of Figure 7.7.

The result of the behavior shown in Figure 7.7 can be a data flow anomaly if $D::f()$ uses variables defined in the state space of D. Because of the order of construction, D's state space will not have been constructed. Whether or not an anomaly exists depends on if default initializations have been specified for the variables used by $f()$. Furthermore, a fault will likely occur if the assumptions or preconditions of $D::f()$ have not been satisfied prior to construction.

In an **anomalous construction behavior fault**, **version 2** (**ACB2**), the constructor of an ancestor class C calls a locally defined polymorphic method $f()$. A data flow anomaly can occur if $f()$ is overridden in a descendant class D and if that overriding method uses state variables inherited from C. The anomaly occurs if the state variables used by $D::f()$ have not been properly constructed by $C::f()$. This depends on the set of variables used by $D::f()$, the order in which the variables in the state of C are constructed, and the order in which $f()$ is called by C's constructor. Note that it is not generally possible for the programmer of class C to know in advance which version of $f()$ will actually execute, or on which state variables the executing version depends. Thus, invoking polymorphic method calls from constructors is unsafe and introduces non-determinism into the construction process. This is true of both ACB2 and ABC1.

In an **incomplete (failed) construction fault** (**IC**), the object's initial state is undefined. In some programming languages, the value of the variables in the state space of a class before construction is undefined. This is true, for example, in C++ but not in Java. Constructors establish the initial state conditions and the state invariant for new instances of the class. To do so, the constructor will generally have statements that define every state variable. In some circumstances, again depending upon the programming language, default or other explicit initializations may be enough. In either case, by the time the constructor has finished, the state of the instance should be well defined. There are two ways for faults to occur. First, the construction process may have assigned an incorrect initial value to a particular state variable. Second,

Table 7.3. IC: Incomplete construction of state variable *fd*

```
Class abstract AbstractFile
{
   FileHandle fd;

   abstract public open();

   public read() {fd.read ( . . . ); }

   public write() {fd.write ( . . . ); }

   abstract public close();
}
```

```
Class SocketFile extends AbstractFile
{
   public open()
   {
      fd = new Socket ( . . . );
   }

   public close()
   {
      fd.flush();
      fd.close();
   }
}
```

the initialization of a particular state variable may have been overlooked. In this case, there is a data flow anomaly between the constructor and each of the methods that will first use the variable after construction (and any other uses until a definition occurs).

An example of incomplete construction is shown by the code fragment in Table 7.3. Class *AbstractFile* contains the state variable *fd* that is not initialized by a constructor. The intent of the designer of *AbstractFile* is that a descendant class provide the definition of *fd* before it is used, which is done by method *open*() in the descendant class *SocketFile*. If any descendant that can be instantiated defines *fd*, and no method uses *fd* before the definition, there is no problem. However, a fault will occur if either of these conditions is not satisfied.

Observe that while the designer's intent is for a descendant to provide the necessary definition, a data flow anomaly exists within *AbstractFile* with respect to *fd* for methods *read*() and *write*(). Both of these methods use *fd*, and if either is called immediately after construction, a fault will occur. Note that this design introduces nondeterminism into *AbstractFile* since it is not known at design time what type of instance *fd* will be bound to, or if it will be bound at all. Suppose that the designer of *AbstractFile* also designed and implemented *SocketFile*, as also shown in Table 7.3. By doing so, the designer has ensured that the data flow anomaly that exists in *AbstractFile* is avoided by the design of *SocketFile*. However, this still does not eliminate the problem of nondeterminism and the introduction of faults since a new descendant can fail to provide the necessary definition.

In a **state visibility anomaly fault** (**SVA**), the state variables in an ancestor class *A* are declared private, and a polymorphic method *A::m*() defines *A::v*. Suppose that *B* is a descendant of *A*, and *C* of *B*, as depicted in Figure 7.8(a). Further, *C* provides an overriding definition of *A::m*() but *B* does not. Since *A::v* has private visibility, it is not possible for *C::m*() to properly interact with the state of *A* by directly defining *A::v*. Instead, *C::m*() must call *A::m*() to modify *v*. Now suppose that *B* also overrides *m* (Figure 7.8(b)). Then for *C::m*() to properly define *A::v*, *C::m*() must call *B::m*(), which in turn must call *A::m*(). Thus, *C::m*() has no direct control over the data flow anomaly! In general, when private state variables are

Figure 7.8. SVA: State visibility anomaly.

present, the only way to be sure to avoid a data flow anomaly is for every overriding method in a descendant to call the overridden method in its ancestor class. Failure to do so will quite possibly result in a fault in the state and behavior of *A*.

Testing Inheritance, Polymorphism and Dynamic Binding

Data flow testing can be applied to OO software by extending the concept of coupling from Chapter 2, section 2.4.2. Recall that a *coupling-def* is a last definition (*last-def*) in one method that can reach a first use, called a *coupling-use*, in another method. A *coupling path* between two program units is a path from a coupling-def to a coupling-use. The path must be def-clear.

In programs that use inheritance, polymorphism, and dynamic binding, identifying the definitions, uses and couplings is more complex, thus the semantics of the OO language features must be considered very carefully.

In the following definitions, *o* is an identifier whose type is a *reference to an instance* of an object, pointing to a memory location that contains an instance (value) of some type. The reference *o* can only refer to instances whose actual instantiated types are either the base type of *o* or a descendant of *o*'s type. Thus, in the Java statement `A o = new B();`, *o*'s base type, or *declared* type, is *A* and it's instantiated type, or *actual* type, is *B*. *B* must be a descendant of *A*.

Figure 7.9 shows a type family rooted at *W*. All members of a type family share some common behavior, as defined in the ancestors of each class. Every type definition by a class defines a type family. Members of the family include the base type of a hierarchy and all types that are descendants of that base type. Figure 7.9(b) shows the four type families defined by the hierarchy in Figure 7.9(a).

(a)

Base Type	Type Family
W	{ W, X, Y, Z }
X	{ X, Y }
Y	{ Y }
Z	{ Z }

(b)

Figure 7.9. Sample class hierarchy (a) and associated type families (b).

The hardest part of data flow analysis of OO programs is the static nondeterminism that polymorphism and dynamic binding introduce. Polymorphism allows one call to refer to multiple methods, depending on the actual type of the object reference, and dynamic binding means that we cannot know which method is called until execution.

An object instance o is considered to be *defined* (assigned a value) when one of the state variables v of the object is defined. An *indirect definition*, or *i-def*, occurs when a method $m()$ defines v. Similarly, an *indirect use* (*i-use*) occurs when $m()$ references the value of v.

When finding the indirect definitions and uses that can occur at call sites through object references, we not only have to consider the syntactic call that is made, but the set of methods that can potentially execute, the *polymorphic call set* (PCS) defined previously. Luckily, **the set of potential methods is finite and can be determined statically**. This analysis uses the term satisfying set.

> *Definition 7.52 Satisfying Set:* For a polymorphic call to method $m()$ through an object reference o, the set of methods that override $m()$, plus $m()$ itself.

Figure 7.10 is based on Figure 7.9. Assume that W includes a method *FactoryForW*() that returns an instance of W. Figure 7.10(a) shows a control flow fragment with an instance of W bound to o. This is a local definition of the object reference o that results from the call to the method *FactoryForW*(). The table in Figure 7.10(b) shows that $W.m()$ defines v, so an indirect definition occurs at node 2 through $o.m()$. Thus, any call to $m()$ with respect an instance of W bound to o results in an indirect definition of the state the object bound to o. There are no indirect uses by $m()$. The table in Figure 7.10(b) also shows all of the indirect definitions and uses that can occur for any instance that is a member of the type family defined by W. Node 3 contains no defs, but an *i-use* of v.

(a)

Indirect definitions and uses for type family *W*		
Method	**Definitions**	**Uses**
W.m	$\{W_r.v\}$	
W.n		$\{W_r.v\}$
X.m	$\{W_r.v, X_r.x\}$	
Y.m	$\{W_r.v, X_r.w\}$	
Y.l		$\{W_r.w\}$
Z.m	$\{W_r.v\}$	
Z.n		$\{W_r.v\}$

(b)

Figure 7.10. Control flow graph fragment (a) and associated definitions and uses (b).

The satisfying set for the call to $m()$ at node 2 is $\{W.m(), X.m(), Y.m(), Z.m()\}$ and the *i-def* set contains the following ordered pairs:

$$i_def(2, o, m()) = \{(W.m(), \{W.v\}), (X.m(), \{W.v, X.x\}), (Y.m(), \{W.v, Y.w\}), (Z.m(), \{W.v\})\}$$

Each pair of definitions and uses in Figure 7.10 indicates a satisfying method for a method and the set of state variables that the method defines. In this example, $X.m()$ defines state variables v from class W and x from X.

From Figure 7.10(b), the *i-use* set for node 2 is empty, as none of the satisfying methods for $m()$ reference any state variable. However, there are two methods that satisfy the call to $o.n()$ at node 3 that have nonempty *i-use* sets (but their *i-def* sets are empty), which yields the following *i-use* set:

$$i_use\ (3, o, n()) = \{(W.n(), \{W.v\}), (Z.n(), \{W.v\})\}$$

Analyzing Polymorphic Paths

Def-use pairs and coupling paths are more complicated in object-oriented programs. In the following definitions, $m()$ is a method, V_m is the set of variables that are referenced by $m()$, and N_m the set of nodes in $m()$'s control flow graph. Also, *defs(i)* is the set of variables defined at node i and *uses(i)* is the set of variables used. *entry(m)* is the entry node of method $m()$, *exit(m)* is the exit node, *first(p)* is the first node in path p, and *last(p)* is the last node.

The following definitions handle the effects of inheritance and polymorphism. The set of classes that belong to the same type family specified by c is *family(c)*, where c is the base ancestor class. *type(m)* is the class that defines method $m()$ and *type(o)* is the class c that is the declared type of variable o. o must refer to an instance of a class that is in the type family of c. *state(c)* is the set of state variables for class c, either declared in c or inherited from an ancestor. *i-defs(m)* is the set of variables

Figure 7.11. Def-use pairs in object-oriented software.

that are indirectly defined within $m()$ and *i-uses(m)* is the set of variables used by $m()$.

A coupling sequence is a pair of nodes that call two methods in sequence; the first method defines a variable and the second uses it. Both calls must be made through the same instance object. The calling method is called the *coupling method* $f()$, and it calls $m()$, the *antecedent method*, to define x, and $n()$, the *consequent method*, to use x. This is illustrated in Figure 7.11 (repeated from Chapter 2). If the antecedent or consequent method is the same as the coupling method, that is a special case that is handled implicitly. If the antecedent or consequent method is called from another method that is called by $f()$, this an *indirect data flow* (right side of Figure 7.11), which we do not discuss.

The *control flow schematic* shown in Figure 7.12 illustrates the coupling method calling both the antecedent and consequent methods. The schematic abstracts away the details of the control flow graph and shows only nodes that are relevant to coupling analysis. The thin line segments represent control flow and the thicker lines indicate control flow that is part of a coupling path. The line segments can represent multiple sub-paths. A path may be annotated with a *transmission set* such as $[o, o.v]$, which contains variables for which the path is definition-clear.

Assuming that the intervening sub-paths are def-clear with respect to the state variable $o.v$, the path in Figure 7.12 from h to i to j to k and finally l forms a *transmission path* with respect to $o.v$. The object o is called the *context variable*.

Every coupling sequence $s_{j,k}$ has some state variables that are defined by the antecedent method and then used by the consequent method. This set of variables is the *coupling set* $\Theta^t_{s_{j,k}}$ of $s_{j,k}$ and is defined as the intersection of those variables defined by $m()$ (an *indirect-def* or *i-def*) and used by $n()$ (an *indirect use* or *i-use*) through the instance context provided by a context variable o that is bound to an instance of t. Which versions of $m()$ and $n()$ execute is determined by the actual type t of the instance bound to o. The members of the coupling set are called *coupling variables*.

Coupling sequences require that there be at least one def-clear path between each node in the sequence. Identifying these paths as parts of complete sequences of nodes results in the set of *coupling paths*. A coupling path is considered to *transmit* a def of a variable to a use.

Figure 7.12. Control flow schematic for prototypical coupling sequence.

Each path consists of up to three sub-paths. The *indirect-def sub-path* is the portion of the coupling path that occurs in the antecedent method $m()$, extending from the last (indirect) definition of a coupling variable to the exit node of $m()$. The *indirect-use sub-path* is the portion of the consequent method $n()$ that extends from the entry node of $n()$ to the first (indirect) use of a coupling variable. The *transmission sub-path* is the portion of the coupling path in the coupling method that extends from the antecedent node to the consequent node, with the requirement that neither the value of the coupling variable nor the context variable is modified.

Each coupling sequence has a single set of coupling paths for each type of *coupling sub-path*. These sets are used to form coupling paths by matching together elements of each set. The set of coupling paths is formed by combining elements of the *indirect-def sub-path* set with an element from the *transmission sub-path* set, and then adding an element of the *indirect-use sub-path* set. The complete set of coupling paths is formed by taking the cross product of these sets.

To see the effects of inheritance and polymorphism on paths, consider the class diagram shown in Figure 7.13(a). The type family contains the classes A, B, and C. Class A defines methods $m()$ and $n()$ and state variables u and v. Class B defines method $l()$ and overrides A's version of $n()$. Likewise, C overrides A's version of $m()$. Definitions and uses for each of these methods are shown in Figure 7.13(b).

Figure 7.14 shows coupling paths for a method that uses the hierarchy in Figure 7.13(a). Figure 7.14(a) shows the declared type of the coupling variable o is A, and Figure 7.14(b) shows the antecedent and consequent methods when the actual type is also A. The coupling sequence $s_{j,k}$ extends from the node j where the antecedent method $m()$ is called to the call site of the consequent method at node k. As shown, the corresponding coupling set for $s_{j,k}$ when o is bound to an instance

(a)

Method	Defs	Uses
A.m ()	{ A.u, A.v }	
A.n ()		{ A.v }
B.n ()		{ A.u }
B.l ()		{ A.v }
C.m ()	{ A.u }	

(b)

Figure 7.13. Sample class hierarchy and *def-use* table.

of A is $\Theta^{A}_{s_{j,k}} = \{A.v\}$. Thus, the set consists of the coupling paths for $s_{j,k}$ that extend from node e in $A.m()$ to the exit node of in $A.m()$, back to the consequent node k in the coupling method, and through the entry node of $A.n()$ to node g. There is no coupling path with respect to $A.u$ because $A.u$ does not appear in the coupling set for $A.m()$ and $A.n()$.

Now, consider the effect on the elements that comprise the set of coupling paths when o is bound to an instance of B, as shown in Figure 7.14(c). The coupling set for this case is different from when o was bound to an instance of A. This is because B provides an overriding method $B.n()$ that has a different use set than the overridden method $A.n()$. Thus, the coupling set is different with respect to the antecedent method $A.m()$ and the consequent method $B.n()$, yielding $\Theta^{B}_{s_{j,k}} = \{A.u\}$. In turn, this results in a different set of coupling paths. The set of coupling paths now extends from node f in $A.m()$ back through the call site at node k in the coupling method, and through the entry node of $B.n()$ to node g of $B.n()$.

Finally, Figure 7.14(c) shows the coupling sequence that results when o is bound to an instance of C. First, observe that execution of the node j in the coupling method results in the invocation of the antecedent method, which is now $C.m()$. Likewise, execution of node k results in the invocation of the consequent method $n()$. Since C does not override $m()$ and because C is a descendant of B, the version of $n()$ that is invoked is actually $B.n()$. Thus, the coupling set for $s_{j,k}$ is taken with respect to the antecedent method $C.m()$ and the consequent method $B.n()$, which yields $\Theta^{C}_{s_{j,k}} = \{A.u\}$. The corresponding coupling path set includes the paths that begin at node e in $C.m()$ and extend to the exit node of $C.m()$, then back to node j of the coupling method, and through the entry node of $B.n()$ to node g, also in $B.n()$.

Table 7.4 summarizes the coupling paths for the examples shown in Figure 7.14. Paths are represented as sequences of nodes. Each node is of the form *method(node)*, where *method* is the name of the method that contains the node, and *node* is the node identifier within the method. Note that the prefixes "call" or "return" are appended to the names of nodes that correspond to *call* or *return* sites.

Figure 7.14. Coupling sequence: *o* of type *A* (a) bound to instance of *A* (b), *B* (c) or *C* (d).

To account for the possibility of polymorphic behavior at a call site, the definition of a coupling sequence must be amended to handle all methods that can execute. A *binding triple* for a coupling sequence contains the antecedent method $m()$, the consequent method $n()$, and the set of coupling variables that result from the binding of the context variable to an instance of a particular type. The triple matches a pair of methods $p()$ and $q()$ that can potentially execute as the result of executing the antecedent and consequent nodes j and k. Each may be from different classes that are members of the type family defined by c, provided that $p()$ is an overriding method for $m()$ or $q()$ is an overriding method for $n()$. There will be exactly one binding triple for each class $d \in family(c)$ that defines an overriding method for either $m()$ or $n()$.

Table 7.4. Summary of sample coupling paths.

Type	Coupling Path
A	$\langle A.m(e),\ A.m(exit),\ f(j.return),\ f(k.call),\ A.n(entry),\ A.n(g)\rangle$
B	$\langle A.m(e),\ A.m(exit),\ f(j.return),\ f(k.call),\ B.n(entry),\ B.n(t)\rangle$
C	$\langle C.m(s),\ C.m(exit),\ f(j.return),\ f(k.call),\ B.n(entry),\ B.n(t)\rangle$

A coupling sequence induces a set of binding triples. This set always includes the binding triple that corresponds to the antecedent and consequent methods, even when there is no method overriding. In this case, the only member of the binding triple set will be the declared type of the context variable, assuming the type is not abstract. If the type is abstract, an instance of the nearest concrete descendant to the declared type is used.

As an example, the set of binding triples for the coupling sequence $s_{j,k}$ shown in Figure 7.14 is shown in Table 7.5. The first column gives the type t of the context variable of $s_{j,k}$, the next two columns are the antecedent and consequent methods that execute for a particular t, and the final column gives the set of coupling variables induced when the context variable is bound to an instance of t. The type hierarchy corresponding to the coupling type t is shown in Figure 7.13.

The instance coupling paths above do not allow for polymorphic behavior when the actual type differs from the declared type. This requires that an instance coupling results in one path set for each member of the type family. The number of paths is limited by the number of overriding methods, either defined directly or inherited from another type. The polymorphic coupling paths are formed by considering each binding triple.

Object-Oriented Testing Criteria

The analysis above allows coupling defs and uses to be identified in the presence of inheritance and polymorphism. This information is used to support testing by adapting the data flow criteria from Chapter 2 to define sub-paths in OO programs that must be tested.

The data flow criteria in Chapter 2 are adapted for inheritance and polymorphism as follows. In the definitions, $f()$ represents a method being tested, $s_{j,k}$ is a coupling sequence in $f()$, where j and k are nodes in the control flow graph of $f()$, and $T_{s_{j,k}}$ represents a set of test cases created to satisfy $s_{j,k}$.

Table 7.5. Binding triples for coupling sequence from class hierarchy in Figure 7.13.

t	p	q	S
A	A.m()	A.n()	{A.v}
B	A.m()	B.n()	{A.u}
C	C.m()	B.n()	{A.u}

The first criterion is based on an assumption that each coupling sequence should be covered during integration testing. Accordingly, *All-Coupling-Sequences* requires that every coupling sequence in $f()$ be covered by at least one test case.

Definition 7.53 All-Coupling-Sequences (ACS): For every coupling sequence s_j in $f()$, there is at least one test case $t \in T_{s_{j,k}}$ such that there is a coupling path induced by $s_{j,k}$ that is a sub-path of the execution trace of $f(t)$.

ACS does not consider inheritance or polymorphism, so the next criterion includes instance contexts of calls. This is achieved by ensuring there is at least one test for every class that can provide an instance context for each coupling sequence. The idea is that the coupling sequence should be tested with every possible type substitution that can occur in a given coupling context.

Definition 7.54 All-Poly-Classes (APC): For every coupling sequence $s_{j,k}$ in method $f()$, and for every class in the family of types defined by the context of $s_{j,k}$, there is at least one test case t such that when $f()$ is executed using t, there is a path p in the set of coupling paths of $s_{j,k}$ that is a sub-path of the execution trace of $f(t)$.

The combination $(s_{j,k}, c)$ is feasible if and only if c is the same as the declared type of the context variable for $s_{j,k}$, or c is a child of the declared type and defines an overriding method for the antecedent or consequent method. That is, only classes that override the antecedent or consequent methods are considered.

ACS requires that coupling sequences be covered but does not consider the state interactions that can occur when multiple coupling variables may be involved. Thus some definitions or uses of coupling variables may not be covered during testing.

The next criterion addresses these limitations by requiring that every last definition of a coupling variable v in an antecedent method of $s_{j,k}$ reaches every first use of v in a consequent method of $s_{j,k}$. Thus, there must be at least one test case that executes each feasible coupling path p with respect to each coupling variable v.

Definition 7.55 All-Coupling-Defs-Uses (ACDU): For every coupling variable v in each coupling $s_{j,k}$ of t, there is a coupling path p induced by $s_{j,k}$, such that p is a sub-path of the execution trace of $f(t)$ for at least one test case $t \in T_{s_{j,k}}$.

APC requires multiple instance contexts to be used, and ACDU requires definitions to reach uses. The final criterion merges these requirements. In addition to inheritance and polymorphism, the *All-Poly-Coupling-Defs-Uses* criterion requires that all coupling paths be executed for every member of the type family defined by the context of a coupling sequence.

Definition 7.56 All-Poly-Coupling-Defs-Uses (APCDU): For every coupling sequence $s_{j,k}$ in method $f()$, for every class in the family of types defined by the context of $s_{j,k}$, for every coupling variable v of $s_{j,k}$, for every node m that has a last definition of v and every node n that has a first use of v, there is at

least one test case t such that when $f()$ is executed using t, there is a path p in the coupling paths of $s_{j,k}$ that is a sub-path of the trace of $f()$.

7.2 TESTING WEB APPLICATIONS AND WEB SERVICES

The use of the world wide Web to deploy software introduces a number of interesting issues for testers to solve. First, an essential difference is that of *deployment* – web applications are deployed on a web server, and made available to any client on the Internet by sending requests through the hypertext transfer protocol (HTTP). The stateless nature of HTTP and the distributed client/server structure creates a unique environment for applications to exist.

Web applications are accessible from virtually anywhere in the world. This factor alone creates a myriad of issues. There is the potential for a variety of users, and they can have different geographic locations, demographics, time zones, disabilities, languages, etc. Web applications are also very competitive, which imposes very high reliability requirements. Users expect web applications to work correctly every time, and if a web application does not, the users will look for a competing web application that does work. This makes testing crucial.

Web applications are also built in novel ways. First and foremost, they are composed of relatively small software components that are distributed (often across multiple computers), run concurrently, and share memory in novel ways, if at all. The HTTP is "stateless," which means that each request/response interaction from client to server and back is independent of the other. Therefore, any state in a web application must be explicitly managed by the software through technologies such as *cookies*, *session objects*, and offline storage such as databases.

Web applications are also created with a multitude of technologies, most fairly new. The technologies used include servlets, Java server pages, ASPs, C-sharp, Java, JavaBeans, XML, Javascript, Ajax, PHP, and many others. Testing the individual components is not much different from testing traditional software, but we are not sure how to test the interactions among these multiple technologies. Moreover, Web applications are usually composed of large numbers of small components that are integrated in novel ways.

The issues for testing can be divided into three broad categories:

1. Testing static hyper text web sites
2. Testing dynamic web applications
3. Testing web services

For this book, a *web page* contains HTML content that can be viewed in a single browser window. A web page may be stored as a static HTML file, or it may be dynamically generated by software such as a Java Server Page, Servlet, or Active Server Page. A *web site* is a collection of web pages and associated software elements that are related semantically by content and syntactically through links and other control mechanisms. A *static web page* is unvarying and the same to all users, and is usually stored as an HTML file on the server. A *dynamic web page* is created by a program on demand, and its contents and structure may be determined by previous inputs from the user, the state on the web server, and other inputs such as

the location of the user, the user's browser, operating system, and even the time of day. A *web application* is a full software program that is deployed across the web. Users access web applications using HTTP requests and the user interface typically executes within a browser on the user's computer (HTML). A *test case* for a web application is described as a sequence of interactions between components on clients and servers. That is, they are paths of transitions through the web application.

7.2.1 Testing Static Hyper Text Web Sites

Early work in testing web sites focused on client-side validation and static server-side validation of links. An extensive listing of existing web test support tools is on a web site maintained by Hower.[2] Available commercial and free tools include link checking tools, HTML validators, capture/playback tools, security test tools, and load and performance stress tools.

These are all static validation and measurement tools. Such testing looks for dead links, that is, links to URLs that are no longer valid, and evaluates the navigation structure to look for invalid paths among pages and shortcuts that users may want.

A common way to model static web sites is as a graph, with web pages as nodes and links as edges. The graph can be built by starting with an introductory page, then recursively performing a breadth-first search of all links from that page. The resulting web site graph is then tested by traversing every edge in the graph (edge coverage).

7.2.2 Testing Dynamic Web Applications

One of many challenges with testing web applications is that the user interface (on the client) and most of the software (on the server) are separated. The tester usually does not have access to data, state, or source on the server. This section first discusses client-side testing strategies and their limitations, then server-side testing strategies that can be used when the tester has access to the implementation.

Client-Side Testing of Web Applications

Testing static hypertext links works well when all pages and links are statically encoded in HTML, but not when parts of the web site are created dynamically or include user inputs. We need some way to generate inputs for form fields. Generating a web site graph also becomes undecidable if some pages or links are only available after certain inputs are provided.

One method is to nondeterministically explore "action sequences," starting from a given URL. Data for form fields can be chosen from inputs pre-supplied by the testing.

Another method for generating input data is based on gathering data from previous users of a web application. This is called *user session data*. Most web servers either capture the data that users submit to web applications on the server, or their settings can be modified to gather the data. These data are collected in name-value pairs and used to generate test inputs.

Another approach to finding inputs is called *bypass testing*. Many web applications impose constraints on the inputs through HTML forms. These constraints come in two forms. Client-side script validation users small programs, usually written in JavaScript, that run on the client's computer and check the input data syntactically before sending it to the server. This is commonly used to ensure that required fields are filled out, numeric data fields only contain numbers, and the like. Another form uses explicit attributes associated with HTML form fields. For example, a text box can be set to only allow strings up to a preset maximum length, and the values from drop-down lists are preset to be the values contained in the HTML. Bypass testing creates inputs that intentionally violate these validation rules and constraints, then submits the inputs directly to the web application without letting the web page validate them.

A limitation of both these approaches comes from the fact that finding all the screens in a web application is undecidable. They rely on heuristic searches to try to identify all screens, but if some screens only appear when certain rare inputs are provided, it may be very difficult to find them. Server-side approaches may have the ability to look at program source, and therefore find more potential screens.

Server-Side Testing of Web Applications

Web software applications allow changes in the control of execution of the application that do not appear in traditional software. In traditional programs, the control flow is fully managed by the program, so the tester can only affect it through test inputs. Web applications do not have this same property. When executing web applications, users can *break* the normal control flow without alerting the "program controller." The model of program controller that is still taught in basic programming and operating system classes does not exactly apply to web applications because the flow of control is distributed across the client and one or more servers. Users can modify the expected control flow on the client by pressing the back or refresh buttons in the browser or by directly modifying the URL in the browser. These interactions introduce unanticipated changes in the execution flow, creating control paths cannot be represented with traditional techniques such as control flow graphs. Users can also directly affect data in unpredictable ways, for example, by modifying values of hidden form fields. Furthermore, changes in the client-side configuration may affect the behavior of web applications. For example, users can turn off cookies, which can cause subsequent operations to malfunction.

This analysis leads to a number of new connections, which we categorize as follows.

- Traditional *static links* are represented in HTML with the `<A>` tag.
- *Dynamic* `<A>` *links* make a request from a static web page to software components to execute some process. No form data is sent in the request, and the type of the HTTP request is always `get`.
- *Dynamic form links* make a request from a form in a static web page by sending data to software components that process the data, using a `<FORM>` tag. The type of HTTP request can be either `get` or `post`, as specified in the `<method>` attribute of the `<FORM>` tag. The data that is submitted via forms impacts the back-end processing, which is important for testing.

- *Dynamically created HTML* is created by web software components, which typically responds to the user with HTML documents. The contents of the HTML documents often depend on inputs, which complicates analysis.
- *State-dependent, dynamically created GUIs* are HTML pages whose contents and form are determined not just by inputs, but by part of the state on the server, such as the date or time, the user, database contents, or session information. The HTML documents can contain Javascript, which are of the web application program that execute on the client. They can also contain links, which determine the execution of the program. The Javascript and links may be different at different times, which is how different users see different programs.
- *Operational transitions* are introduced by the user outside of the control of the HTML or software. Operational transitions include use of the back button, the forward button, and URL rewriting. This type of transition is new to web software, very difficult to anticipate, and often leads to problems because they are difficult for programmers to anticipate.
- *Local software connections* are among back-end software components on the web server, such as method calls.
- *Off-site software connections* occur when web applications access software components that are available at a remote site. They can be accessed by sending calls or messages to software on another server, using HTTP or some other network protocol. This type of connection, while powerful, is difficult to analyze because the tester does not know much about the off-site software.
- *Dynamic connections* occur when new software components are installed dynamically during execution. Both the J2SE platform and .NET allow web application to detect and use the new components. This type of connection is especially difficult to evaluate because the components are not available until **after** the software is deployed.

The net result of these types of transitions is that traditional analysis structures such as control flow graphs, call graphs, data flow graphs, and data dependency graphs cannot accurately model web applications. That is, the program's *possible* flow of control cannot be known statically. These analysis structures are needed for testing, thus new techniques are needed to model web applications to support these activities.

An early attempt to develop tests for web applications tried to apply data flow analysis to web software components. Definition-use pairs can be split among client web pages and multiple server software components.

An *atomic section* is a section of HTML (possibly including scripting language routines such as JavaScript) that has the property that if part of the section is sent to a client, the entire section is. The atomic section allows web applications to be modeled in the same way that basic blocks and control flow graphs allow non-web applications to be modeled. Such graphs can then be used to implement the graph criteria in Chapter 2.

Thus far, these ideas have only appeared in the research literature, and have not made it to practical application.

7.2.3 Testing Web Services

Web services introduce a few more wrinkles for testers. Unfortunately, the term "web service" has not been standardized and the literature contains several different definitions, some of which conflict. The most common depend on particular technologies such as XML and the simple object access protocol (SOAP). This book will take a more generic approach. A *web service* is a distributed, modular application, whose components communicate by exchanging data in structured formats.

Testing web services is difficult because they are distributed applications with unusual runtime behaviors. The design and implementation details are not available, so testers need to use client-side testing. A single business process often involves multiple web services. Moreover, these multiple web services might be located on different servers, and belong to different companies. Web services interact by passing messages and data in structured ways, specifically with XML. Although technologies exist to verify syntactic aspects of the interactions, the problem of whether the two web services behave correctly with all possible messages is more difficult.

The goal of web service communication is to allow web services to be described, advertised, discovered and invoked through the Internet. This infrastructure uses several technologies to let web services function together. The *extensible markup language* (XML) is used to transmit messages and data. The *universal description, discovery and integration* (UDDI) specification is used to maintain directories of information about web services. These directories record information about web services, including location and requirements, in a format that other web services and applications can read. The *web services description language* (WSDL) is used to describe how to access web services and what operations they can perform. SOAP helps software transmit and receive XML messages over the Internet.

Research into testing web services has just begun and so far has centered on the messages. Inputs to web service components are XML messages that correspond to specific syntactic requirements. These requirements can be described in XML schemas, which provide a grammar-based description of the messages. Researchers are beginning to apply the syntax testing criteria in Chapter 5 to test web services.

7.3 TESTING GRAPHICAL USER INTERFACES

Graphical user interfaces (GUIs) account for half or more of the amount of source code in modern software systems. Yet there is very little help for testing this large amount of software. GUI testing falls into two categories. *Usability testing* refers to assessing how usable the interface is, using principles from user interface design. While usability testing is extremely important, it is out of the scope of this book. *Functional* testing refers to assessing whether the user interface works as intended.

Functional testing can be further broken down into four types. *GUI system testing* refers to performing system testing through the GUI. The only difference in GUI system testing other types of testing is in how to automate the tests. *Regression testing* refers to testing the UI after changes are made. The most common type of regression test tool is a capture-replay tool. Generally speaking, they *capture* some user inputs, *replay* them through the UI after changes have been made, and report the differences. Dozens of capture-replay tools are available on the market,

the concept is fairly simple, so they is not discussed further in this chapter. *Input validation testing* tests how well the software recognizes and responds to invalid inputs. The techniques used for input validation testing are not particularly affected by using a graphical user interface as opposed to other types of interface, thus this topic is not discussed in this section. Finally, *GUI testing* refers to assessing how well the GUI works. The tester asks questions such as "Do all the UI controls work as intended?", "Does the software allow the user to navigate to all screens that the user would expect?", and "Are inappropriate navigations disallowed?".

7.3.1 Testing GUIs

One fairly obvious technique for testing GUIs is to use some sort of finite state machine model, then apply graph-based criteria from Chapter 2. Modeling a GUI as a state machine is fairly straightforward, as they are naturally event-based systems. Every user event (pushing a button, entering text, navigating to another screen) causes a transition to a new state. A path is a sequence of edges through the transitions, and represents one test case. An advantage of this approach is that the expected output is simply the final state that the test case input should arrive in. The problem with this approach is the possibility of state explosion; even small GUIs will have thousands of states and transitions.

The number of states can be reduced in one of several ways. *Variable finite state machines* reduce the number of abstract states by adding variables to the model. These models must be created by hand. Also, if these models are used for verification via an automated test oracle, effective mappings between the machine's abstract states and the GUI's concrete state need to be developed by hand.

A variation on the state-machine model for GUI test case generation partitions the state space into different machines based on user tasks. The tester identifies a user task (called a *responsibility*, that can be performed with the GUI. Each responsibility is converted to a *complete interaction sequence* (CIS). These are similar, but not exactly the same as the use cases in Chapter 2. Each CIS is a graph, and graph coverage criteria can be used to cover them. Although it is relatively simple to define the responsibilities, converting them into FSM models must be done by hand, which is a significant expense.

Another approach to testing GUIs relies on modeling the user behavior, specifically by mimicking *novice users*. The intuition is that expert users take short, direct paths through the GUI that are not particularly useful for testing. Novice users, on the other hand, take indirect paths that exercise the GUI in different ways. In this approach, an expert starts by generating a few sequences of inputs. These initial sequences are used to generate tests by applying genetic algorithms to modify them to look more like sequences created by novices.

A more recent compromise approach is based on an *event-flow* model through the GUI. The event flow model proceeds in two steps. First, each event is encoded in preconditions. *Preconditions* include the state in which the event can be executed and the *effects* of the event, that is, the state change that occurs as a result of the event. Second, the tester represents all possible sequences of events that can be executed on the GUI as a set of directed graphs. Third, preconditions and effects are used to generate tests by using a goal-directed approach. Because the expected

outputs are the target states of the events, this allows test oracles to be automated created and checked. The directed graph model is used to generate tests that satisfy criteria in Chapter 2.

More details on how these methods can be applied can be found in the papers in the bibliographic section.

7.4 REAL-TIME SOFTWARE AND EMBEDDED SOFTWARE

Real-time software systems must respond to externally generated input stimuli within a finite and specified period. Real-time systems are often embedded and operate in the context of a larger engineering system, sometimes designed for a dedicated platform and application.

Real-time systems typically interact with other sub-systems and processes in the physical world. This is called the *environment* of the real-time system. For example, the environment of a real-time system that controls a robot arm includes items coming down a conveyor belt and messages from other robot control systems along the same production line. Real-time systems usually have explicit time constraints that specify the response time and temporal behavior of real-time systems. For example, a time constraint for a flight monitoring system can be that once landing permission is requested, a response must be provided within 30 seconds. A time constraint on the response time of a request is called a *deadline*. Time constraints come from the dynamic characteristics of the environment (movement, acceleration, etc.) or from design and safety decisions imposed by a system developer.

Timeliness refers to the ability of software to meet time constraints. For example, a time constraint for a flight monitoring system can be that once landing permission is requested, a response must be provided within 30 seconds. Faults in the software can lead to software timeliness violations and costly accidents. Thus testers need to detect violation of timing constraints.

Real-time systems are sometimes called *reactive* because they react to changes in their environment. These changes are recognized by *sensors*, and the system influences the environment through *actuators*. Since real-time systems control hardware that interacts closely with entities and people in the real world, they often need to be dependable.

A *real-time application* is defined by a set of tasks that implements a particular functionality for the real-time system. The execution environment of a real-time application is all the other software and hardware needed to make the system behave as intended, for example, real-time operating systems and I/O devices.

Two types of real-time tasks are commonly used. *Periodic* tasks are activated at a fixed frequency, thus all the points in time when such tasks are activated are known beforehand. For example, a task with a period of 4 time units will be activated at times 0, 4, 8, etc. *Aperiodic* tasks can be activated at any point in time. To achieve timeliness in a real-time system, aperiodic tasks must be specified with constraints on their activation pattern. When a constraint is present, the tasks are called *sporadic*. A common constraint is a *minimum inter-arrival time* between two consecutive task activations. Tasks may also have an *offset* that denotes the time before any instance may be activated.

Testers often want to know the *longest execution time* (the literature usually calls this "*worst-case*." Unfortunately, this is very difficult to estimate, so a common goal

in testing is to cause the software to execute as long as possible; hopefully reaching the worst-case execution time.

The *response time* of a real-time task is the time it takes from when it is activated until it finishes its execution. The response times of a set of concurrent tasks depend on the order in which they are scheduled to execute. This is called the *execution order* of tasks.

Issues with Testing Real-Time Systems

A landmark paper by Schütz described issues that need to be considered when testing real-time systems. In the context of real-time software, *observability* is the ability to monitor or log the behavior of a real-time system. Observability is usually increased by inserting *probes* that reveal information about the current state and internal state-changes in the system. A problem is that probes in real-time software can influence the temporal behavior of the system, so removing them can invalidate the test results. This problem is usually referred to as the *probe-effect*. The probe-effect problem is usually solved by leaving the probes in the software, but directing their output to a channel that consumes the same amount of resources but is inaccessible during operation.

A special type of probe is a built-in software (or hardware) component that monitors the activity in the system and then leave that component in the system. In systems with scarce computing resources, the probe-effect makes it desirable to keep the amount of logging to a minimum. This is fairly common in real-time embedded systems when the hardware is severely constrained (such as a spaceship).

Two related concepts are reproducibility and controllability. *Reproducibility* is when the system repeatedly exhibits identical behavior when stimulated with the same inputs. Reproducibility is a very desirable property for testing, particularly during regression testing and debugging.

It is very difficult to achieve reproducibility in real-time systems, especially in event-triggered and dynamically scheduled systems. This is because the actual behavior of a system depends on elements that have not been expressed explicitly as part of the systems input. For example, the response time of a task depends on the current load of the system, varying efficiency of hardware acceleration components, etc. These systems are nondeterministic. A high degree of controllability is typically required to effectively test nondeterministic software.

If the software under test is nondeterministic and has low controllability, testers must use statistical methods to ensure the validity of test results. This is usually done by executing the same test inputs many times to achieve statistically significant results. A minimum requirement on controllability is that a sequence of timed inputs can be repeatedly injected in the same way.

Timeliness Faults, Errors, and Failures

The term *timeliness fault* denotes a mistake in the implementation or configuration of a real-time application that may result in incorrect temporal behaviors. For example, a timeliness fault can be that a condition in a branch statement is wrong and causes a loop in a task to iterate more times than expected. Another example is when two tasks disturb each other (for example, via unprotected shared hardware and software resources) in a unanticipated way. Both these examples of timeliness faults may case some part of a task to execute longer than expected. Another type

of timeliness fault occurs when the environment (or sensors and actuators) behaves differently than expected. For example, if an interrupt handling mechanism is subject to an unforeseen delay, then the internal inter-arrival time may become shorter than expected.

A *timeliness error* occurs when the system internally deviates from assumptions about its temporal behavior. This is similar when a non-real-time program has an internal state error. Timeliness errors are difficult to detect without extensive logging and precise knowledge about the internal behavior of the system. In addition, timeliness errors might only be detectable and lead to system level timeliness failures for specific execution orders.

A *timeliness failure* is a violation of a time constraint that can be observed externally. In a hard real-time system, this has some penalty or consequence for the continued operation of the overall system. Since time constraints typically are expressed with respect to the externally observable behavior of a system (or component), timeliness failures are often easy to detect.

Testing for Timeliness

Test criteria must be adapted to address timeliness because it is difficult to characterize a critical sequence of inputs without considering the effect on the set of active tasks and real-time protocols. However, the test criteria as presented in earlier chapters seldom use information about real-time design in test case generation, nor do they predict what execution orders may reveal faults in off-line assumptions.

Timeliness is traditionally analyzed and maintained using scheduling analysis techniques or timeliness is regulated online through admission control and contingency schemes. However, these techniques make assumptions about the tasks and activation patterns that must be correct for timeliness to be maintained. Further, a full schedulability analysis of non-trivial system models is complicated and requires specific rules to be followed by the run-time system. Testing for timeliness is more general; it applies to all system architectures and can be used to gain confidence in assumptions by systematically sampling among the execution orders that can lead to missed deadlines. However, only some of the possible execution orders reveal timeliness violations in the presence of timing faults. Therefore, the challenge is to find execution orders that will cause timeliness faults to result in failure.

Real-time testing often depends on formal models of the software. One approach is to describe the software with timed Petri nets. Then graph criteria such as edge or path coverage can be used to try to find test inputs that will violate timeliness constraints.

Another approach to modeling timeliness is to specify time constraints in a *constraint graph*, and specify the system under test using process algebra. Only constraints on the system inputs are considered.

Another approach specifies time constraints using a *clock region graph*. A timed automation specification of the system is then "flattened" to a conventional input output automation that is used to derive conformance tests for the implementation in each clock region.

Another modeling technique that is used for real-time systems is *temporal logic*. The elements of test cases are pairs of timed input and outputs. These pairs can be combined and shifted in time to create a large number of partial test cases;

the number of such pairs grows quickly with the size and constraints on the software.

Timed automata have also been used to verify sequences of timed action transitions. This approach uses a reachability analysis to determine what transitions to test; that is, a graph-based approach. This can suffer from state space explosion for large dynamic models. One way to ameliorate that problem is to use a sampling algorithm based on grid-automata and nondeterministic finite-state machines to reduce the test effort.

A non-formal model approach uses genetic algorithms. Data are gathered during execution of the real system and visualized for post analysis. Fitness of a test case is calculated based on its uniqueness and what exceptions are generated by the systems and test harness during test execution.

One approach to generating tests is to statically derive execution orders of a real-time system before it is put into operation. Each execution order is treated as a separate sequential program, and conventional test methods can be applied. This only works if all task activation times are fixed.

Most of the above approaches are based on graph criteria (Chapter 2) in one way or another. A different approach is based on mutation (Chapter 5). In mutation-based timeliness testing, potential faults are modeled as mutation operators. Mutants that have the potential to violate timeliness are identified, and test cases are constructed that try to kill the mutants.

Eight types of mutants have been defined. The **task set mutation operator** changes the points in time when a resource is taken. The **execution time mutation operator** increases or decreases the execution time of a task by a constant time delta. The **hold time shift mutation operator** changes the interval of time a resource is locked. The **lock time mutation operator** increases or decreases the time when a resource is locked. The **unlock time mutation operator** change when a resource is unlocked. The **precedence constraint mutation operator** adds or removes precedence constraint relations between pairs of tasks. The **inter-arrival time mutant operator** decreases the inter-arrival time between requests for a task execution by a constant time Δ. The **pattern offset mutation operator** changes the offset between two such patterns by a constant Δ time units. Test cases to kill timeliness mutants are created by model-checking and by genetic algorithms.

7.5 BIBLIOGRAPHIC NOTES

The bibliographic notes in this chapter follow the order of the sections above, object-oriented software, web applications, GUIS, and real-time and embedded software.

Good sources for building abstractions in object-oriented language are by Meyer [241], Liskov, Wing and Guttag [212, 213], and Firesmith [119]. Liskov, Wing and Guttag contributed the substitution principle. Discussions about whether the substitution principle should always be followed can be found by Lalonde and Pugh [199] and Taivalsaari [323].

Binder pointed out how OO relationships tend to be complex [35]. and Berard first articulated the differences in integration [31]. The connection between inheritance, polymorphism, dynamic binding and undecidability is due to Barbey [24].

Doong and Frankl [105] wrote a seminal paper on state-based object-oriented testing. The three class testing levels of *intra-method testing*, *inter-method testing*, and *intra-class testing* are due to Harrold and Rothermel [152]; Gallagher and Offutt [132] added *inter-class testing*.

Several papers focused on inter-method and intra-class testing [118, 152, 281, 315], testing interactions between classes and their users [284] and system-level testing [181].

The yo-yo graph was provided by Alexander and Offutt [266], and based on the discussion from Binder that execution can sometimes "bounce" up and down among levels of inheritance [35]. The categories of OO faults and anomalies are due to Alexander and Offutt [9, 266]. Most of the OO coupling and data flow test criteria were developed by Alexander as part of his thesis work [7, 10, 11].

The idea of using a graph to represent static web sites was initially proposed by Ricca and Tonella [300]. Kung et al. [198, 215] also developed a model to represent web sites as a graph, and provided preliminary definitions for developing tests based on the graph in terms of web page traversals. Their model includes static link transitions and focuses on the client side with only limited use of the server software. They define *intra-object* testing, where test paths are selected for the variables that have def-use chains within the object, *inter-object* testing, where test paths are selected for variables that have def-use chains across objects, and *inter-client* testing, where tests are derived from a reachability graph that is related to the data interactions among clients.

Benedikt, Freire, and Godefroid [30] initiated the idea of "action sequences" in a tool called VeriWeb. VeriWeb's testing is based on graphs where nodes are web pages and edges are explicit HTML links, and the size of the graphs is controlled by a pruning process.

Elbaum, Karre, and Rothermel [112, 113] proposed the idea of "user session data" to generate test cases for web applications.

Liu, Kung, Hsia, and Hsu [216] first tried to apply data flow analysis to web software components. The focus was on data interactions and their model did not incorporate dynamically generated web pages or operational transitions.

Bypass testing is due to Offutt, Wu, Du, and Huang [278]. The analysis about the connections in web applications in Section 7.2.2 is due to Offutt and Wu [359, 360]. They also introduced the concept of an atomic section.

Di Lucca and Di Penta [217] proposed testing sequences through a web application that incorporates some operational transitions, specifically focusing on the back and forward button transitions. Timewise, this paper is the first published work addressing operational transitions, although it postdates the earlier technical report by Offutt and Wu [359]. Di Lucca and Di Penta's model focused on the browser capabilities without considering the server-side software or dynamically generated web pages.

The use of syntax-based testing techniques to create XML messages as tests for web services components is due to Offutt and Wu [279, 280].

Nielson [253] provides an excellent overview of the importance of web usability in general and includes a discussion of usability testing.

Early work in using state-machine models to generate tests for GUIs are by Clarke [79], Chow [77], Esmiloglu [115], and Bernhard [32]. The variable finite state machine model is due to Shehady et al. [312].

The concept of partitioning the GUI state space into responsibilities is due to White et al. [346, 347]. The idea of modeling novice user behavior to generate tests is due to Kasik [183]. The event flow model has been extensively developed by Memon et al. [235, 236, 237, 238, 239, 240, 250].

The literature on testing real-time and embedded software is much smaller than some of the other topics in this book. The concepts presented in this chapter only introduce the topic. The following references are far from complete, but will help introduce the interested reader to the field.

General knowledge on real-time software systems and timeliness can be found in Young [364], Ramamrithaam [296], and Schütz [309]. Many of the issues in testing real-time systems were published by Schütz [310]. This probe-effect is due to Gait [131]. Timeliness was discussed by Verissimo and Kopetz [331].

The method of using Petri nets is due to Braberman et al. [42]. Cheung et al. [68] presented a framework for testing multimedia software, including temporal relations between tasks with "fuzzy" deadlines.

The framework for testing time constraints using constraint graphs and process algebras is due to Clarke and Lee [78].

The clock region graph approach is by Petitjean and Fochal [287]. Krichen and Tripakis [193] addressed limitations in applicability of previous client-side approaches and suggested a method for conformance testing using nondeterministic and partially observable models. Their testing criteria were inspired by Hessel et al. [159].

The temporal logic approach is due to Mandrioli et al. [225], who based their work on SanPietro et al. [308]. The timed automata approach is due to Cardell-Oliver and Glover [61]. Another automata-based approach was by En-Nouaary et al. [114], who introduced the sampling algorithm using grid-automata. Similarly, Nielsen and Skou [252] use a subclass of timed automata to specify real-time applications. Raymond et al. [299] presented a method to generate event sequences for reactive systems.

The genetic algorithm approach is due to Watkins et al. [337]. Morasca and Pezze [245] proposed a method for testing concurrent and real-time systems that uses high-level Petri nets for specification and implementation. The technique of statically deriving execution orders is due to Thane [326] and Pettersson and Thane [288]. Wegener et al. explored the capabilities of genetic algorithms for testing temporal properties of real-time tasks [338], attempting to create inputs that produced the worst and best-case execution times. The application of mutation to timeliness faults was by Nilsson [254, 255, 257, 256].

NOTES

1 Researchers interpret "interesting" to mean having fun problems to solve, but developers, especially managers, should interpret "interesting" as a threat to timely completion of a quality product.

2 The URL is `http://www.softwareqatest.com/qatweb1.html`

8

Building Testing Tools

Test criteria are used in several ways, but the most common way is to evaluate tests. That is, sets of test cases are evaluated by how well they cover a criterion. Applying criteria this way is prohibitively expensive, so automated coverage analysis tools are needed to support the tester. A *coverage analysis tool* accepts a test criterion, a program under test, and a collection of test cases, and computes the amount of coverage of the tests on the program under test. This chapter discusses the design techniques used in these tools. We do not discuss individual tools, although many are available. We also do not discuss the user interface issues, but focus on the core internal algorithms for measuring coverage.

8.1 INSTRUMENTATION FOR GRAPH AND LOGICAL EXPRESSION CRITERIA

The primary mechanism used to measure coverage is instrumentation. An *instrument* is additional program code that does not change the functional behavior of the program but collects some additional information. The instrument can affect the timing in a real-time system, and could also affect concurrency. Thus, such applications require special attention. Careful design can make instrumentation very efficient.

For test criteria coverage, the additional information is whether individual test requirements have been met. An initial example of instrumentation is shown in Figure 8.1. It illustrates a statement that is added to record if the body of an "if-block" has been reached.

8.1.1 Node and Edge Coverage

One of the simplest criteria to instrument for is node coverage. This method most obviously works with node coverage on program source, but the general idea works with arbitrary graphs. Each node in the graph is assigned a unique identifier number.

Original Function	With Instrument

public int min (A, B)
{
  int m = A;
  if (A > B)
  {
  m = B;
  }
  return (m);
}
	public int min (A, B)
	{
	  int m = A;
	  if (A > B)
	  {
	  Mark: "if body has been reached"
	  m = B;
	  }
	  return (m);
	}

Figure 8.1 Initial example of instrumentation.

An array is created that is indexed by the node numbers (called nodeCover []). Next, the following instrument is inserted at each node i: "nodeCover [i]++;."

It is important that the nodeCover[] array be "persistent," that is, it must be saved to disk after each test case. This allows results to be accumulated across multiple test cases. After some tests have been executed, every node i for which nodeCover[i] is zero has not been covered. If nodeCover[i] is not zero, its value represents the number of times node i was reached.

This process is shown in Figure 8.2. The nodeCover[] array must be read before execution starts and is shown on node 1 in the figure. Each node is annotated with the appropriate instrumentation ("nc" is used as an abbreviation for "nodeCover"). An automated coverage analysis tool can insert the instruments at the beginning

Figure 8.2. Node coverage instrumentation.

Figure 8.3. Edge coverage instrumentation.

of basic blocks or in front of individual statements. The latter is less accurate, but simpler to implement, so is fairly common in commercial tools. The instruments can also be inserted in the source code and then compiled separately, into executable files, or into intermediate forms such as Java ByteCode. Java Reflection could also be used to insert instrumentation, although we know of no tool that uses this technique.

Instrumenting for edge coverage is only slightly more complicated than for node coverage. Each edge in the graph is assigned a unique identifier number. An array is created that is indexed by the edge numbers (`edgeCover []`). Next the following instrument is inserted onto each edge i: "`edgeCover[i]++;`."

This process is illustrated in Figure 8.3. As with node coverage, the `edgeCover[]` array must be read before execution starts. Each node in Figure 8.3 is annotated with "`ec`" as an abbreviation for "`edgeCover`." The instrumentation for some edges is sometimes omitted if coverage is implied by another. For example, in Figure 8.3, any test that causes edge 3 to be executed will also cause edge 5 to be executed.

Some structures do not explicitly represent all edges. For example, the following program source does not have a location for the instrumentation on the else edge:

```
if (isPrime)
{  // save it!
   primes[numPrimes] = curPrime;
   numPrimes++;
}
```

Therefore the else clause must be explicitly added to instrument for branch coverage:

```
if (isPrime)
{  // save it!
   primes[numPrimes] = curPrime;
   numPrimes++;
}
else
   edgeCover[5]++;
```

8.1.2 Data Flow Coverage

Data flow coverage is somewhat more complicated than node and edge coverage. The primary difference is that the criterion is based on two locations in the program: one for the definition and one for the use. As with node coverage, each node is assigned a unique number, but two arrays are used. Since edges can also contain uses, each edge must also be assigned a unique number. The technique is to use one array to keep track of definitions and another to keep track of uses.

At each location where variable `x` is defined, the statement "`defCover[x] = i;`" is added, where `i` is the node number. This means that `defCover[x]` will store the last location where `x` was defined. The second array keeps track of uses. For a variable `x`, `useCover[]` stores uses on a node or edge. For each node or edge `i` where a variable `x` is used, the statement "`useCover[i, x, defCover[x]]++;`" is added. The array location `useCover[i, x, defCover[x]]` indicates that `x` is used at node (or edge) `i` and the definition from node `defCover[x]` has reached the use at `i`.

Figure 8.4 illustrates instrumentation for All-uses. Variables `x` and `y` are defined (through parameter passing or assignments) at node 1 and `y` is redefined at node 2. Variable `x` is used at nodes 4 and 5 and `y` is used at node 6. If the path 1, 2,

Figure 8.4. All uses coverage instrumentation.

6 is toured, then when the statement "useCover[6, y, defCover[y]]++;" at node 6 is reached, defCover[y] will be 2, and coverage of DU pair [2, 6] will be recorded. If, however, the path 1, 3, 4, 6 is toured, then defCover[y] will be 1 when node 6 is reached, so coverage of DU pair [1, 6] will be recorded.

All-uses instrumentation is a little hard to grasp at first. If in doubt, the reader should walk through a couple of simple examples to verify that the useCover[] array stores the needed information. After execution, the useCover[] array records zeros for DU pairs that have not been covered, and nonzero values for DU pairs that have been. Other analysis, not covered in this book, is required to determine which definitions can reach which uses, and which DU pairs have def-clear paths. Pairs of definitions and uses without def-clear paths between them will result in zero values in the useCover[] array.

8.1.3 Logic Coverage

We show how to instrument for just one of the logic coverage criteria from Chapter 3. The others are similar.

Instrumenting for the logic coverage criteria requires more instrumentation than structural or data flow instrumentation. The abstract view is whenever a predicate is reached, each clause in each predicate must be evaluated separately to determine which of the test requirements on the predicate have been satisfied. These evaluations are implemented in separate methods that mark special arrays to record which test requirements have been satisfied.

Consider the graph in Figure 8.5(a). The first predicate is (A && B) and it results in the test requirements (F, T), (T, T), and (T, F). Figure 8.5(b) gives the method that is called by the instrumented statement at node 1. It implements the predicate on the edge (1, 2) and marks the array CACCCover[] to indicate which test requirement has been satisfied.

The second predicate, C && (D || E), has three clauses and because of the "or" condition, some choices are possible in the test requirements. This is illustrated in Figure 8.5(c). Again the predicate is evaluated, one clause at a time, and the appropriate test requirement coverage is recorded. Note that CACCCover[5] is recorded in three different places. This represents the fact that one test requirement can be satisfied by one of three clause truth assignments.

A different approach to instrumenting for the ACC criteria is, for each predicate, simply to record the combination of truth values for each clause in that predicate. Such an approach is not appropriate if clauses have side effects, or if predicates rely on short circuit evaluation, such as testing an array index prior to dereferencing the array. However, the advantage of such an approach is that the analysis of satisfaction of the ACC criteria is separated from the code under instrumentation. Hence, the criteria analysis engine can be applied to ACC coverage data collected from any artifact.

8.2 BUILDING MUTATION TESTING TOOLS

Mutation was described in Chapter 5. It is widely regarded as the most difficult criterion to satisfy, and empirical studies have consistently found it to be stronger than

Figure 8.5. Correlated active clause coverage instrumentation.

other criteria in terms of the number of faults it can detect. It is also all but impossible to apply by hand; thus automation is a must. Not surprisingly, automating mutation is more complicated than automating other criteria. Simply adding instrumentation to the program does not work. The literature on building mutation testing systems is large, and we have lots of experience, at least in the research community. And luckily, changes in language design and advances in tools have made this kind of system much easier to build. This section explores issues with building mutation systems and explains how a mutation testing system can be built in a reasonably efficient manner.

The first thing to realize about a mutation testing system is that it is in large part a *language system*. Programs must be parsed, modified and executed. Please refer

to Figure 5.2. To create mutants, the program must first be parsed, and the mutant creation engine must know the language. Likewise, the equivalence detector must be based on the semantics of the language. When the program is run ("Run T on P"), the system must recognize two possibilities that are usually abnormal behavior, but that are in fact normal behavior in mutation testing. If a mutant crashes, that's actually a good thing and the mutation system should mark the mutant as being dead. Similarly, if the mutant goes into an infinite loop, that also means the mutant has failed. The runtime system must handle both of these situations.

The first mutation systems were based on interpreting an intermediate form. This section presents the interpretation architecture, then points out the problems with that approach. The next solution is a compilation architecture, which brings in other problems. A compromise approach, schema-based mutation, is how a mutation tool should be built today.

8.2.1 The Interpretation Approach

In an interpretation architecture, a program under test is first parsed into an intermediate form. This is usually not a standard intermediate form that compilers use, but a special-purpose language designed specifically to support mutation. The create mutants component directly modifies the intermediate form to insert a mutant, and the mutation system includes a special-purpose interpreter. The interpreter can easily handle the bookkeeping when mutants are killed, and can respond to program failure. The usual way to handle infinite loops is first to run a test case on the original program, count the number of intermediate instructions executed, then run the test case on a mutant. If the mutant uses X times more intermediate instructions (X has usually been set at 10), then the mutant is assumed to be in an infinite loop and marked dead.

The interpretive approach has several advantages. Having full control of the execution environment is very helpful. Parsing the program and creating mutants is efficient. Creating mutants by making small changes to the intermediate form is simple and efficient. Individual mutants do not need to be stored on disk, only the rules for changing the intermediate form need to be saved.

A difficulty with this approach is that the mutation system must be a complete language system: parser, interpreter, and run-time execution engine. It is similar to building a compiler, but in many ways more complicated. Thus, building such a system is a significant investment. Another disadvantage is that they run fairly slowly; an interpreted program runs about 10 times slower than a compiled program. Researchers have found that it can take up to 30 minutes to run all mutants on a 30 line program.

8.2.2 The Separate Compilation Approach

The separate compilation approach tries to trade more up-front costs to save time on the backend. Each mutant is created as a complete program by modifying the source of the original program under test. Then each mutant is compiled, linked, and run.

This has the advantage of running mutants much faster than with the interpretive approach. However, it is more difficult to keep track of which mutants have

been killed, and more difficult to handle run-time failures and infinite loops. The separate compilation approach can also suffer from a *compilation bottleneck*, particularly with large programs. This is also a problem with small programs that run very quickly, because the time to compile and link can be much greater than the time to execute. It is also difficult to apply weak mutation with the separate compilation approach.

8.2.3 The Schema-Based Approach

The "schema-based approach" was developed to solve the above problems. Instead of mutating an intermediate form, the MSG, short for Mutant Schema Generation, approach encodes all mutations into one source-level program, called a *metamutant*. The metamutant program is then compiled (once) with the same compiler used during development and is executed in the same operational environment at compiled-program speeds. Because mutation systems based on mutant schemata do not need to provide the entire run-time semantics and environment, they are significantly less complex and easier to build than interpretive systems, as well as more portable. Because of extra computation, MSG systems run slightly slower than compiler systems, but significantly faster than interpretive systems.

Let's look at how MSG works in more detail. A *program schema* is a template. A *partially interpreted program schema* syntactically resembles a program, but contains free identifiers that are called *abstract entities*. The abstract entities appear in place of some program variables, datatype identifiers, constants, and program statements. A schema is created through a process of *abstraction*. A schema can be *instantiated* to form a complete program by providing appropriate substitutions for the abstract entities.

For mutation, a mutant schema is created that uses abstract entities to represent elements in the program that are changed by mutants. A mutant schema has two components, a *metamutant* and a *metamethod set*, both of which are represented by syntactically valid (i.e., compilable) constructs.

As an example, consider the arithmetic operator replacement mutation operator (AOR). If the following statement appears in the program under test:

```
delta = newGuess - sqrt;
```

then it is mutated to create the following seven mutants:

```
delta = newGuess +  sqrt;
delta = newGuess *  sqrt;
delta = newGuess /  sqrt;
delta = newGuess ** sqrt;
delta = newGuess %  sqrt;
delta = newGuess;
delta = sqrt;
```

These mutations can be "generically" represented as

```
delta = newGuess arithOp  sqrt;
```

where *arithOp* is a *metaoperator* abstract entity. This abstraction is implemented by using a metamethod:

```
delta = arithOp (newGuess, sqrt, 44);
```

The method arithOp() performs one arithmetic operation. The third argument, "44," represents the location in the program from where the metamethod is called. The metaprogram accepts a parameter that tells it which mutant to execute. The metamethod checks that global parameter, and if the mutant is an AOR mutant at location 44, it performs the appropriate arithmetic operation, otherwise it returns the original expression (newGuess - sqrt).

8.2.4 Using Java Reflection

An approach that takes advantage of a modern language feature to combine the interpretive and compiler-based approach uses *reflection*. Reflection allows a program to (1) access its internal structure and behavior, and (2) manipulate that structure, thereby modifying its behavior based on rules supplied by another program. Reflection is possible only in languages that support it, most notably Java and C-sharp. Both support reflection by allowing access to the intermediate form, that is, the Java bytecode. Reflection comes in three flavors. *Compile-time reflection* allows changes to be made when the program is compiled. *Load-time reflection* allows changes to be made when the program is loaded into the execution system (JVM with Java). *Run-time reflection* allows changes to be made when the program is executed.

Reflection is a natural way to implement mutation analysis for several reasons. First, it lets programmers extract information about a class by providing an object that represents a logical structure of the class definition. This means the mutation system does not have to parse the program. Second, it provides an API to modify the behavior of a program during execution. This can be used to create mutated versions of the program. Third, it allows objects to be instantiated and methods to be invoked dynamically. Finally, some of the OO operators cannot be implemented via MSG. For example, the hiding variable deletion (HVD) from Chapter 5 requires that declaration of variables that hide an ancestor's variable be deleted. This affects every reference to the variable and thus cannot be implemented in a mutant schema.

Java provides a built-in reflection capability with a dedicated API. This allows Java programs to perform functions such as asking for the class of a given object, finding the methods in that class, and invoking those methods. However, Java does not provide full reflective capabilities. Specifically, Java supports only *introspection*,

which is the ability to introspect data structures, but does not directly support modification of the program behavior. Several reflection systems have been built and are available to support compile-time and load-time reflection. As of this writing, the technology for run-time reflection is still developing.

Bytecode translation is similar to reflection, but uses a different approach. Bytecode translation inspects and modifies the intermediate representation of Java programs, bytecode. Since it directly handles bytecode, it has some advantages over separate compilation and MSG approaches. First, it can process an off-the-shelf program or library that is supplied without source code. Second, it can run at load time, when the Java Virtual Machine loads a class file. Bytecode translation is not as efficient as run-time reflection, but presently the technology is more stable.

8.2.5 Implementing a Modern Mutation System

A modern mutation system would use a combination of MSG for mutants that do not affect the structure of the program, and reflection for mutants that do. Although still more complicated than instrumenting for statement or branch coverage, the amount of programming is significantly less than for interpretive systems. This is the approach taken by the muJava system mentioned in Chapter 5.

8.3 BIBLIOGRAPHIC NOTES

We were not able to find any published references for how to do instrumentation. However, researchers and tool builders have been using techniques such as what we have presented for many years.

PIMS [5, 52, 54, 211], an early mutation testing tool, pioneered the general process typically used in mutation testing of creating mutants (of Fortran IV programs), accepting test cases from the users, and then executing the test cases on the mutants to decide how many mutants were killed.

In 1987, this same process (of add test cases, run mutants, check results, and repeat) was adopted and extended in the Mothra mutation toolset [96, 101, 262, 268], which provided an integrated set of tools, each of which performed an individual, separate task to support mutation analysis and testing. Although other mutation testing tools have been developed since Mothra [95, 97, 220, 330], Mothra is likely the most widely known mutation testing system extant.

Many of the advances in mutation testing addressed the performance cost. They usually follow one of three strategies: *do fewer*, *do smarter*, or *do faster*.

The *do fewer* approaches try to run fewer mutant programs without incurring unacceptable information loss. Mutant sampling [4, 53, 356] uses a random sample of the mutants and is the simplest *do fewer* approach. A 10% sample of mutant programs, for example, was found to be only 16% less effective than a full set in ascertaining fault detection effectiveness. The effects of varying the sampling percentage from 10% to 40% in steps of 5% were later investigated by Wong [356]. An alternative sampling approach is proposed by Şahinoğlu and Spafford [106] that does not use samples of some *a priori* fixed size, but rather, based on a Bayesian sequential probability ratio test, selects mutant programs until sufficient evidence

has been collected to determine that a statistically appropriate sample size has been reached.

Wong and Mathur suggested the idea of *selective mutation* as applying mutation only to the most critical mutation operators being used [230, 357]. This idea was later developed by Offutt et al. [269] who identified a set of selective operators for Fortran-77 with Mothra. Results showed that selective mutation provide almost the same test coverage as nonselective mutation. The operators presented in this Chapter are based on the selective approach.

The use of nonstandard computer architecture has been explored as a *do smarter* approach. This approach distributes the computational expense over several machines. Work has been done to adapt mutation analysis system to vector processors [229], SIMD machines [192], Hypercube (MIMD) machines [75, 275], and Network (MIMD) computers [365]. Because each mutant program is independent of all other mutant programs, communication costs are fairly low. At least one tool was able to achieve almost linear speedup for moderate sized program functions [275].

Weak mutation [167] is another *do smarter* approach. It is an approximation technique that compares the internal states of the mutant and original program immediately after execution of the mutated portion of the program. Experimentation has shown that weak mutation can generate tests that are almost as effective as tests generated with strong mutation, and that at least 50% and usually more of the execution time is saved. The Leonardo system [270, 271], which was implemented as part of Mothra, did two things. It implemented a working weak mutation system that could be compared easily with strong mutation, and evaluated the extent/firm concept by allowing comparisons to be made at four different locations after the mutated component: (1) after the first evaluation of the innermost expression surrounding the mutated symbol, (2) after the first execution of the mutated statement, (3) after the first execution of the basic block that contains the mutated statement, and (4) after **each** execution of the basic block that contains the mutated statement (execution stops as soon as an invalid state is detected).

In another "do smarter" approach, Fleyshgakker and Weiss describe algorithms that improve the run-time complexity of conventional mutation analysis systems at the expense of increased space complexity [120]. By intelligently storing state information, their techniques factor the expense of running a mutant over several related mutant executions and thereby lower the total computational costs.

In the *separate compilation* approach, each mutant is individually created, compiled, linked, and run. The Proteum system [95] is an example of the separate compilation approach. When mutant run times greatly exceed individual compilation/link times, a system based on such a strategy will execute 15–20 times faster than an interpretive system. When this condition is not met, however, a *compilation bottleneck* [75] may result.

To avoid compilation bottlenecks, DeMillo, Krauser, and Mathur developed a *compiler-integrated* program mutation scheme that avoids much of the overhead of the compilation bottleneck and yet is able to execute compiled code [97]. In this method, the program under test is compiled by a special compiler. As the compilation process proceeds, the effects of mutations are noted and *code patches* that represent these mutations are prepared. Execution of a particular mutant requires only that the appropriate code patch be applied prior to execution. Patching is

inexpensive and the mutant executes at compiled speeds. However, crafting the special compiler needed turns out to be very expensive and difficult.

Untch developed a new execution model for mutation, the *m*utant *s*chema *g*eneration (MSG) method [330]. A *program schemata* is one complete, compilable, program that encode all mutants into one *metaprogram*. This is the current state of the art for mutation [277].

The OO mutation operators were designed to test the essential OO language features of inheritance, polymorphism, and dynamic binding [11, 219, 281]. The mutation operators used in muJava developed through several research papers by Kim, Clark, and McDermid [184, 185], Chevalley and Thévenod-Fosse [69, 71], Alexander et al. [8, 266], and Ma, Kwon, and Offutt [219, 220, 221]. MuJava uses OpenJava [324, 325] for compile-time reflection because it provides enough information to generate mutants and it is easy to use.

9

Challenges in Testing Software

We end this book with a discussion of three challenging areas in testing software. Although researchers have been interested in emergent properties for many years, the area, far from being "solved," continues to escalate in terms of importance for industry. Likewise, testability is attracting renewed attention due to characteristics of some of the newer software technologies. Finally, we suggest some directions for software testing in practice and in the research arena.

9.1 TESTING FOR EMERGENT PROPERTIES: SAFETY AND SECURITY

Testing for emergent properties presents special challenges. This section offers high level guidance for engineers faced with testing systems where safety and/or security play an important role.

Emergent properties arise as a result of collecting components together into a single system. They do not exist independently in any particular component. Safety and security are classic emergent properties in system design. For example, the overall safety of an airplane is not determined by the control software by itself, or the engines by themselves, or by any other component by itself. Certainly, the individual behavior of a given component may be extremely important with respect to overall safety, but, even so, the overall safety is determined by the interactions of all of these components when assembled into a complete airplane. In other words, an airplane engine is neither safe nor unsafe considered by itself because an airplane engine doesn't fly by itself. Only complete airplanes can fly, and hence only complete airplanes can be considered safe or unsafe with respect to flying. Likewise, the security of a web application is not determined by the security of a back-end database server by itself, or by a proxy server by itself, or by the cryptographic systems used by themselves, but by the interactions of all of these components.

Not only are systems where safety and/or security are important far more common than might first appear, they are becoming more common for a variety of technical and social reasons. For example, consider security for a system for controlling heating and air conditioning. A traditional system that can only be controlled through physical access cannot be compromised remotely. However, the ability to

control several such systems from a central remote location clearly is cheaper, and using the Internet as the means of connecting such systems to a controller is an easy implementation. Unfortunately, such an approach also means that the system now has a nonzero risk of malicious attack from anywhere in the world. Further, if the developers of the central controller don't consider security during the initial design of the system, it is unlikely that the resulting product will be sufficiently secure, even if significant effort is put into security "upgrades" later.

More generally, there are three basic themes driving the pervasiveness of safety and security issues. The first, used in the example above, is *connectivity*. The Internet is everywhere, and there are powerful incentives for many applications to take advantage of "free" connectivity. The second is *complexity*. Networked, distributed software is hard to build well, and even harder to assess as being well built. The third reason is *extensibility*. Applications get ever more moving parts, some of which appear "on-the-fly," which means that they aren't even well-defined when a test engineer carries out an assessment.

It is important to distinguish testing safety and security functions from testing for safety or security. The former is, in principle, no different than testing any other sort of function. The latter focuses on undesirable, possibly malicious, behavior. For example, if a control system has an *emergency shutdown* feature, intended for use in situations an operator determines are hazardous, then this is a safety function, and the test engineer evaluates its functionality in the same way as any other system functionality. For an example in the security context, consider a function that authenticates a user with a id/password scheme. The test engineer assesses the authentication function in the same way as any other system functionality. In contrast, suppose there exist situations where the emergency shutdown should be invoked, but isn't. Or suppose that someone can become an authenticated user without invoking the authentication function. Upon reflection, the reader should realize that these two latter problems are *much* harder to address because they force the analyst to address the negative argument of showing that something bad thing won't happen, rather than the positive argument that something good will happen. These latter two examples illustrate the problem of testing for, respectively, safety and security, and this type of problem occupies our attention for the remainder of this section.

Roughly speaking, a system is *safe* if it reasonably free of unacceptable hazards, and a system is *secure* if it is reasonably robust with respect to malicious threats. The literature certainly offers valuable refinements on these rough definitions, but they are not appropriate for the level of material presented in this section. It is important to understand that insisting on completeness with respect to safety or security is hopeless. Rather, the process proceeds by identifying and ranking hazards and threats, choosing requirements that address these hazards and threats in a satisfactory way, and then selecting designs and implementations that promise to meet the requirements. Sometimes this process fails in the sense that it proves impossible to complete without unduly compromising safety or security. Arguably, such systems are better left unbuilt.

As everyone knows, the process outlined above is idealized and often not followed in practice. Unfortunately, the test engineer faced with the task of testing such a system will find it necessary to carry out the process anyway – or else have

no idea what he or she is testing for or why. Worse, testing safety or security "into" a system after the fact is generally hopeless, precisely because of the emergent nature of these properties. Stated differently, the properties are not, by definition, in the components, and, if not provided for by design, will only be in the system by accident. Sadly, relying on blind luck rarely works out well.

Complicating matters is the difficult technical nature of many safety and security requirements. In the case of safety, quantitative requirements may be so stringent that they exceed the bounds of engineering practice. Consider the oft-cited requirement on safety critical software in commercial aircraft, namely a failure rate of 10^{-9} per 10 hour flight. Butler and Finelli wrote a great paper explaining why meeting such "ultra" requirements is infeasible, no matter what development and testing technology, current or future, is used. Instead of attempting to meet infeasible quantitative requirements, the common approach is to use qualitative "safety cases." Not surprisingly, such qualitative approaches rely heavily on sound development processes. Although the safety side is hard, the security side is harder, since it is complicated by the clever nature of human beings. Malicious human behavior is extremely hard to anticipate, much less thwart.

So, how does one go about testing such systems?

The first lesson to draw from the discussion above is that the test engineer has to have a clear idea of what to test for. Explicit documentation of safety and security requirements, along with links to the hazards and threats they address is crucial for making this determination. Some safety and security requirements are untestable. For example, consider the problem of "backdoor" code inserted during development. System testing has no chance of finding such code, because the malicious designer has so many options for choosing how to open the back door. In other words, spending testing resources on this threat is a poor expenditure of resources. Instead, the threat must be addressed through process, personnel practices, or inspections.

A more profitable place to spend testing resources is in probing assumptions underlying the safety and security model. Every design relies on such assumptions because the assumptions go all the way back to the ranking of hazards and threats. For example, designers of the Arianne 4 rocket design made assumptions about the positions, velocities, and accelerations that the rocket would experience early in flight, and these assumptions were used in the placement of exception handlers in the control code. These assumptions were valid for the Arianne 4, but, regrettably not for the Arianne 5, where much of the control software was reused. From a testing perspective, access to such assumptions is extremely valuable, which means that documenting these assumptions is a critical process.

Assumptions are a fertile source of tests for security as well. Consider the common issue of downloading software from a network onto a device. One strategy is to inform the user of the impending download (and its inherent risk), and allow the user to accept or decline. This strategy incorporates at least three assumptions: first, that users will be able to make an informed decision; second, that users will be willing to make an informed decision; and third, that it will be possible for the user to actually say "No." Many cell phones rely on the third assumption, but it is easily violated in an environment where the malicious adversary can query the user at a very high rate. Specifically, if a mobile phone has a virus on it that queries nearby mobile phones about downloading itself, and responds to "No" by immediately re-asking

the question, then users are subject to a classic denial of service (DOS) attack, and may eventually click "Ok" out of frustration. The phone is then infected with the virus. Notice that the security problem here is not fixable, or even well defined, at the level of the component phone, but is instead a classic emergent aspect of the phone, the phone network, and the population of users, including malicious users. Further, addressing the problem requires a fundamental reassessment of the rules under which mobile phones should be allowed to interact with each other.

Reuse of software is a classic area where prior assumptions may be violated, and hence testing is appropriate. Consider web software. In a typical web application, business logic embedded in the web server protects back-end applications such as databases from malicious user inputs. But if the web application is redeployed to a web service, the web server may be removed, and the back-end applications are once again vulnerable.

9.1.1 Classes of Test Cases for Emergent Properties

The intent of the prior discussion is to describe some basic issues in testing emergent properties. Here we switch gears and provide some common strategies for selecting test cases.

1. **Develop misuse cases**. Use cases are a common modeling tool that enumerate expected usage of the software, typically during the requirements analysis phase. Misuse cases are the same, except they explore unintended usage. The requirements phase is the ideal time to consider such scenarios, since it is the correct place to eliminate or mitigate them.
2. **Identify assumptions and then devise test cases that violate them**. Even after misuse case analysis, assumptions of varying plausibility will be left over. As discussed above, testing these can be extremely productive.
3. **Identify configuration issues and design tests to check them**. Configuration issues, in which inconsistent versions of components are used to realize a system, is a ripe area for problems. Configuration tests tend to be oriented more to the development process than the product itself, except for the case where the components in a product are explicitly aware of their configuration data.
4. **Develop invalid input tests**. Invalid inputs are an extremely fertile area for security testing. Simple invalid input attacks such as buffer overflow and query injection attacks are responsible for a great many security breaches. Every time a system collects data from its environment, the data can violate assumptions about its size or content. Fortunately, generating test data to probe the limits on input data is easy to automate and well understood using the techniques developed in Chapter 5.

Another complicating factor in assessing safety and security is the difference between developers and users. Naturally, systems are built by developers, and hence it is developers who make provisions for system safety and system security. However, systems are employed by users, typically operations specialists and network administrators. Developers and users usually have very different skill sets, levels of training, and mental models of the application domain. Developers focus on software

artifacts, particularly code. Conversely, users tend to be only vaguely aware that there is software in their systems, and instead focus on the abstractions suitable for their jobs. We need systems that are safe and secure, not only in theory when conceived by developers, but also in practice when employed by users. Hence, test engineers should bias their perspective toward the usage environment, as opposed to the development environment, when considering tests for safety and security.

9.2 SOFTWARE TESTABILITY

Software *testability* is an important notion distinct from software testing. In general, software testability is an estimate or measurement of a conditional probability, namely, assuming that a given software artifact contains a fault, how likely is it that testing will reveal that fault. We are all familiar with software development projects where, despite extensive testing, faults continue to be found. Testability gets to the core of how easy or hard it is for faults to escape detection – even from well-chosen test suites.

There are a variety of reasons to study testability. Given testability estimates for a set of software artifacts, the test engineer has a number of options:

1. For artifacts with low testability, the test engineer knows that testing alone is unlikely to result in a satisfactory job of verifying that the artifact meets his requirements, and so the test engineer can pursue alternative verification means. In the case of critical software, this might mean formal analysis. In more ordinary software, design or code reviews might be more appropriate.
2. Artifacts with low testability can be altered in a variety of ways that improve testability. Although we will defer the technical details until later in this section, any approach that improves testability will almost certainly be worth the investment.
3. The test engineer can proceed to test artifacts with high testability and be confident that the test results are accurate indicators of artifact quality.
4. In cases where testability is low and resistant to improvement, and where alternative means of verification are impractical, the test engineer has a solid argument to take to management that the artifact poses an undue amount of risk.

Various authors have proposed different approaches for tackling testability. Some authors have focused on software observability and software controllability as the key components of testability. We begin with a model of testability built on the fault, error, and failure model from Chapter 1.

We define testability in terms of three attributes of the RIP model from Chapter 1: reachability, infection, and propagation. We assume that if an artifact contains a fault, then that fault must exist at some location in the artifact. For the fault at a given location to result in a failure, three things must happen. First, execution must reach the location of the fault. Second, the execution of the fault must result in an infection of the program state, an error, at the given location. Finally, the error must propagate from the location to an output of the artifact. The *sensitivity* of a given location is simply the probability that reachability, infection, and propagation will all occur for that location.

Notice that we do not specify what a fault at a given location necessarily looks like. The reader might think it helpful to think of either average case or worst case behavior for all faults possible at that location. The *testability* of an artifact is defined as the minimum value of the sensitivity over all locations in the artifact.

To clarify the idea that sensitivities are probabilities, it is necessary to model inputs as being drawn from one or more *usage distributions*. In other words, we assume that test cases are sampled from some profile. In practice, defining usage distributions is quite easy to do, and such an approach to testing has numerous proponents; citations are available in the bibliographic notes section.

Given a distribution for selecting test inputs, measuring the reachability probability for any given location is a direct affair. Further, if the reachability probability for a given location is deemed to be too low, it is easy to change the input distribution to channel a larger fraction of the test inputs to that location. Stated more directly, the reachability attribute of testability is under the direct control of the test engineer. One way to think of this property is in terms of controllability. Software with low controllability will probably suffer from low reachability probabilities, no matter how the input distribution is skewed.

Estimating infection probabilities is a bit trickier, since it requires the adoption of both a fault model and an infection model, both of which must be suitable for the location in question. One choice is to use mutation analysis for the fault model, and measure infections by observing the effect of mutants on candidate program states.

Determining propagation probabilities also involves a certain amount of modeling. One choice is to use perturbation models to alter the program state at the location in question and then observe whether the infection persists to the outputs. One way to think of this property is in terms of observability. Software with low observability will probably suffer from low propagation. However, there are quite a few things that can be done to re-engineer code to have higher propagation.

First, there is the dimensionality of the output space to consider. For example, consider a method with a single boolean output. Such a method is prone to have low testability, if, for no other reason, than propagation from any infection is limited by the collapsing of the entire internal state to a single boolean output. The test engineer can interact with system designers to see if additional outputs are possible, thereby giving better access to the internal state, and hence better propagation.

Second, there is the notion of assertion checking. Assertion mechanisms are ideal for transforming internal state inconsistencies into observable events, typically via an exception mechanism. Designers can assist this process by providing extensive checks of internal states. Such checks need not be enabled during operational deployment; from the testability perspective, it is sufficient that they be enabled during testing only.

9.2.1 Testability for Common Technologies

Object-oriented software presents special challenges for testability. The chief reason is that objects encode state information in instance variables, and access to these instance variables is usually indirect. Consider the simple example of a stack. The method `push(Object item)` changes the state of the stack, perhaps by storing `item` in an instance variable `Object[] elements`. The method `top()` yields access to the most

recent element in the stack, and so a push() followed immediately by a top() allows the test engineer to verify that the element returned by top() is indeed the most recent element. However, access to older elements is hidden; the typical stack interface does not, and should not, allow direct access to these elements. From an observability perspective, the stack interface is a problem for the test engineer. Most classes are far more complex than the stack class, so it should come as no great surprise that testability for object-oriented software tends to be low.

Inheritance compounds the lack of testability for object-oriented software. To continue the stack class example, suppose a sub-class logs the number of push() calls. To implement the sub-class, the typical programmer overrides the push() method to increment the log and then calls the superclass push() method to take care of the state change. Such modifications of classes are routine in object-oriented development. From the observability perspective, testability for the subclass suffers from the fact that some of the updated instance variables are not even in the sub-class being tested. In fact, it is quite likely that the developer of the subclass doesn't have access to the source code for the superclass.

To improve observability, there are two basic approaches. The first is to require developers to provide additional get methods that allow the test engineer access to the full state. Although this may be a reasonable approach in cases where extensive test sets are developed concurrently with, or even prior to, the class itself, it is quite likely that the test engineer will still end up having to test code where, for whatever reason, the software was not developed in accordance with testability guidelines.

A second approach is a tool approach that exploits something like the Java reflection mechanism to access internal variables, independent of whether source code is available. On the other hand, this approach suffers from the fact that interpreting the data values so captured is a nontrivial task for the test engineer.

Maintenance for tests is also an issue, since every time the developers change the implementation of a class, tests that access internal state of that class are unlikely to execute at all, let alone execute correctly.

Web applications pose a different set of challenges for testability. Again, both controllability and observability are likely to be extremely low for a typical web application. To appreciate the situation, consider the architecture of a typical web server, in which a proxy interacts with an individual client, a session manager oversees the entire pool of clients, application logic determines processing of client requests, and data flows in and out of a back-end database. From the client perspective, almost all of this infrastructure is intended to be invisible, hence accessing much of the state is impossible from the client. The server pieces are quite likely to be distributed, not only across multiple hardware platforms, but also across multiple corporate organizations. Bringing high testability to such an environment is still a research topic.

9.3 TEST CRITERIA AND THE FUTURE OF SOFTWARE TESTING

When the authors started their careers in the mid 1980s, the field of software engineering was very different from today. A huge difference can be expressed in terms of economics. In the 1980s, the economics of software engineering were such that the cost of applying high-end testing criteria usually exceeded the economic benefits.

This meant that software development organizations had very little incentive to test their software well.

This situation had a lot to do with how software was marketed and deployed. Most software came "bundled" on the computer – the cost of the software was included with the computer and customers had very few choices. Other software was purchased "shrink-wrapped," that is, it was bought in a sealed package and customers bought and paid for the software without having a chance to try it. Another major mechanism for marketing software was by "contract," where a customer contracted a supplier to provide some specific software. By far the biggest purchaser at this time was the US Government, led by the US Department of Defense. Although there have always been attempts to hold software suppliers accountable for quality, the system had numerous problems. From a broad brush point of view, when problems were found, the supplier could often get additional funding to repair the software. The fact that there are few standards for testing or quality assurance, and the requirements often changed frequently during development, made it very difficult to hold suppliers responsible for quality problems or cost overruns.

Of course, these points do not cover the entire field. We have always needed software that had to be highly reliable, such as safety-critical software. However, this part of the market was tiny in the 1980s and there was not enough critical mass to support tool vendors or widespread education in software testing.

The bottom line is that for much of our careers, the software market has been dominated by one supplier and one purchaser. This has resulted in a very noncompetitive market, and relatively little motivation for quality assurance or testing. That is, software testers have been "selling" techniques that are seldom needed.

On a more positive note, the field has dramatically changed. The software market is much larger; it is more competitive, there are significantly more users, and we are using software in many more applications. A major impetus for this change has been the development of the Web. The World Wide Web provides a different way to **deploy** software. Instead of buying a computer with software bundled inside, or buying a shrink-wrapped CD from a store, or hiring programmers to build custom-made software, users can now run software that is deployed on a server across the Web. This makes the market much more competitive – if users are unhappy with software on one web site, they can easily "vote with their mouse" by going to a different site. Another aspect of the Web is that more software is available to more users. Such a growth in the market brings a correspondingly increase in expectations. Highly educated, technically proficient, users will tolerate lots of problems. The "blue screen of death" presents a familiar situation to an engineer: the system went into an error state and we need to reboot. However, the excise tasks of rebooting and other compensating actions are not well received by the broader user community. When software is made available to billions of users who **did not explicitly purchase the software**, those users have very low tolerance for errors. How successful would Amazon or Netflix be if they sent the wrong product, sent the product to the wrong place, billed the customers incorrectly, or simply froze during use? Efficiency and time-to-market matter less on the Web and in E-Commerce applications. The success of Google relative to earlier search engines is a well-known examples of this fact, but it is true for many smaller businesses as well. As we say in our classes, on the Web "it is better to be late and better than early and worse."

There are other reasons why quality and reliability requirements continue to go up in software. In the 1980s we had a few embedded software applications. Most were high-end, special-purpose, and very expensive. The devices and the software were built by specialists, often in the military or avionics industry. We are now literally surrounded by embedded software. Companies who make mobile phones can view themselves not as vendors of phones, but as software companies who happen to put their software on a particular kind of device. Many other electronic devices include substantial software: PDAs, portable music players, cameras, watches, calculators, TV and Internet cable boxes, home wireless routers, home appliance remote controllers, garage door openers, refrigerators, and microwave ovens. Even the humble toaster now comes with software-controlled sensors. New cars are thick with sensors and software – doors that open automatically, sensors for airbags, automatic seat movement, sensors to detect when tire pressure is low, and automatic parking. All of this software has to work very well. When we embed software in an appliance, users expect the software to be **more reliable** than the appliance, not less.

Of course, the elephant in this particular room has become security. At one time, security was all about clever algorithms to encrypt data. Later security became a database problem, then a networking problem. Most software security vulnerabilities today are **due to software faults**. Software must be highly reliable to be secure, and any software that uses a network is vulnerable.

How can we develop software for these many applications that work as reliably as we need it to? One part of the answer is that software developers need more testing and more efficient testing. Most of the criteria and techniques in this book have been known for years. The time is ripe for these concepts to be taught in computer science and software engineering curriculum, incorporated into more and better quality software testing tools, and most importantly, adopted by industry.

9.3.1 Going Forward with Testing Research

This book presents a different way to look at software. Instead of considering software testing as being conducted differently with different criteria throughout the development process (that is, unit testing is different from module testing is different from integration testing is different from system testing), this book presents the view that test criteria are defined on four models of software (graphs, logical expressions, input spaces, and syntax), and the models can be developed from any software artifact. That is, a graph can be created from units (methods), from modules, from integration information, or from the system as a whole. Once the graph is created, the same criteria are used cover the graph, no matter where it came from.

This book presents a total of 36 criteria on the four structures. Many other, closely related, criteria appear in the literature, but were not included in this book because we felt they would be unlikely to be used in practice. Much of the testing research in the last 30 years has focused on inventing new test criteria. Our opinion is that the field does not need many more criteria. It does need several other things, however.

A continuing need will be research into engineering existing criteria for new technologies and situations. Chapter 7 illustrates this by showing how the criteria in Chapters 2 through 5 can be engineered to apply to object-oriented software, web

applications, graphical user interfaces, and real-time, embedded software. Computing technology will continue to evolve, presenting new and interesting challenges to testing researchers. We hope that the criteria presentation in this book will provide a foundational basis for this kind of research.

A major problem for development managers is deciding which testing criteria to use. In fact, this question delays adoption of improved test technologies, particularly if the cost (real or perceived) of adoption is high. Each of Chapter 2 through 5 included a subsumption graph that indicated which criteria subsumed which other criteria. For example, if we have satisfied edge coverage on a graph, we have provably satisfied node coverage. This information is only part of the question, however, and a small part for practical software developers. Assuming "subsumes" means "better" in the sense of possibly finding more faults, a development manager wants to know "how much better?" and "how much more expensive?". Although dozens, and perhaps hundreds, of papers have been published presenting experimental comparisons of test criteria, we still are far from having enough knowledge to tell a development manager which criterion to use and when.

An expansion on this question is the question of how to compare test criteria that **cannot** be compared with subsumption. This kind of comparison must be experimental in nature, and almost certainly will need to be replicated. For example, data flow criteria (Chapter 2) and mutation criteria (Chapter 4) are incomparable with subsumption. Several papers were published in the 1990s that compared program mutation with one or more of the data flow criteria. Individually, none of the papers was convincing, yet taken together, they have convinced researchers that mutation testing will find more faults. However, none of this research looked at large-scale industrial software, and none had much useful information about cost (beyond the observation that mutation usually requires more tests).

Another major issue is how best to fully automate these criteria. Most of the early publications on individual criteria, and a lot of the followup papers included algorithms, tool development, and efficiency improvements. Chapter 8 provides an introduction to tool building but it is important to note that many of these concepts have not made it out of the research laboratory. Commercial test tools must be more robust, more efficient, and have much better user interfaces than research tools. Although companies have tried to sell criteria-based test tools for decades, it is only recently that companies such as Agitar and Certess have had commercial success. The earlier point that testing is become more crucial to the economic success of software companies should mean that the market for software test tools is in growth mode.

A significant problem with adopting any new technology in practice is how it affects the current development process. Making it easy for developers to integrate test techniques into their existing process is a key ingredient for success. If test tools can be integrated into compilers (as with early test tools such as dead code detection) or with IDEs like Eclipse (such as Agitar does), then the tools will become much more accessible to developers.

A major outstanding problem with automation is test data generation. Automatic test data generation has been a subject of research since the early days, but useful commercial tools are still few and far between. This is not because of lack of interest, but because it is a very hard problem. Some of the earliest research papers

in software testing focused on automatic test data generation, but progress proved so difficult that most researchers diverted into more tractable problems. Some early work focused on randomly generating test inputs. While simpler to implement than other approaches, random test data generation has two problems. The first is that, as Dick Lipton said, "faults ain't fish." That is, faults are not distributed uniformly (randomly) through the program like fish in a school. Any programmer knows that faults tend to clump around troublesome portions of the program or difficult to define regions in the input space. The second problem with random test data generation is with structure data. It is easy to randomly generate numbers and characters, but when the data has some structure (addresses, product records, etc.), random generation becomes much harder.

The most complicated idea is to analyze program source and use detailed information about the software to automatically generate test data specifically to satisfy test criteria. The most common analysis method has been symbolic evaluation, often coupled with constraint representations and sometimes slicing. Some analysis techniques have been static and others dynamic. Solving the equations or constraints for actual input values has been done with exact solutions such as constraint solving (linear programming) and inexact special purpose heuristics. Significant problems with analysis-based techniques are the so-called "internal variable problem," loops, and indeterminate memory references such as pointers, arrays and aliases. The internal variable problem refers to the fact that we may have specific requirements on test inputs such as "*X must be greater than Y on line 5306*," yet *X* and *Y* are not inputs. Finding input values to indirectly control the values of *X* and *Y* has been troublesome. Dynamic analysis techniques are the most promising way to address these problems, however, these problems are why analysis-based techniques are mostly considered to apply to small structures such as program units.

The most recent automatic test data generation approach uses search techniques such as genetic algorithms. Search-based approaches have several advantages, including being easy to implement and being very flexible. They cannot use as much information as the analysis-based methods, and thus can get stuck on values that are hard to find. For example, if a search-based technique is used to achieve edge coverage on a method-level control flow graph, a predicate such as `x+y = z` presents difficulty for search-based techniques. This problem is compounded if several such predicates have to be satisfied to reach a portion of the method. For this reason, search-based techniques have so far achieved more success at the system level than the unit level.

9.4 BIBLIOGRAPHIC NOTES

An excellent, if slightly dated, place to start on software for safety critical systems is Leveson's book [207]. Butler and Finelli wrote a great article on the futility of assuring ultra high confidence levels in software via *any* means, including testing [59]. Heimdahl [155] presents a good state-of-the-art perspective on the challenges of safety in software intensive systems. McGraw [233] is a good current source for the many issues to consider in building secure software. For a somewhat older treatment, readers might wish to try Rubin [306]. Denning's book [103] is a classic, although it is out of print and hence harder to obtain.

Casting the testability problem in terms of controllability and observability comes originally from hardware testing; Friedman [129] and Binder [34] adapted it to software testing. Voas [333] developed the three part sensitivity model described in this chapter. Applying the testability model in conduction with random testing yields an interesting verification argument [334].

Other criteria that we chose to omit are cataloged in a recent book by Pezze and Young [289]. Other references can be found in the research papers cited in previous chapters.

A good start for how the Web changes the way software is viewed is a book by Powell [292]. Several authors have commented on the inadequacies of subsumption for practical use [149, 340, 344]. We know of four papers that compared data flow with mutation testing [125, 230, 267, 274]. Lots of other papers have appeared that empirically compared other test criteria [25, 49, 81, 123, 124, 135, 258, 302, 304, 311, 327].

Random automatic test data generation approaches date back to the mid-1970s [36, 176, 218, 243]. Older specification-based test data generation approaches use formal specifications [23, 76]; more recent approaches use model-based languages such as the UML [2, 46, 70]. Grammar-based test data generation approaches appeared more than 35 years ago and are currently being revived for XML-based software such as web services [150, 231, 248, 280]. Several researchers investigated how to use program analysis techniques for automatic test data generation [39, 80, 101, 102, 174, 190, 191, 267, 295, 342]. The use of search-based techniques, primarily genetic algorithms started with informally acquiring knowledge about tests [351] and has been refined since then [179, 234, 242].

List of Criteria

Criterion Name	Acronym	Page Defined
Chapter 2		
Node Coverage	NC	33
Edge Coverage	EC	34
Edge-Pair Coverage	EPC	35
Prime Path Coverage	PPC	35
Simple Round Trip Coverage	SRTC	36
Complete Round Trip Coverage	CRTC	36
Complete Path Coverage	CPC	36
Specified Path Coverage	SPC	36
All-Defs Coverage	ADC	48
All-Uses Coverage	AUC	48
All-du-Paths Coverage	ADUPC	48
Chapter 3		
Predicate Coverage	PC	106
Clause Coverage	CC	106
Combinatorial Coverage	CoC	107
General Active Clause Coverage	GACC	109
Correlated Active Clause Coverage	CACC	109
Restricted Active Clause Coverage	RACC	110
General Inactive Clause Coverage	GICC	112
Restricted Inactive Clause Coverage	RICC	112
Implicant Coverage	IC	139
Unique True Point Coverage	UTPC	140
Corresponding Unique True Points Near False Point Coverage	CUTPNFP	142

Criterion Name	Acronym	Page Defined
Chapter 4		
All Combinations Coverage	ACoC	160
Each Choice Coverage	ECC	160
Pair-Wise Coverage	PWC	161
T-wise Coverage	TWC	161
Base Choice Coverage	BCC	162
Multiple Base Choice Coverage	MBCC	162
Chapter 5		
Terminal Symbol Coverage	TSC	172
Production Coverage	PDC	172
Derivation Coverage	DC	172
Mutation Coverage	MC	175
Mutation Operator Coverage	MOC	175
Mutation Production Coverage	MPC	175
Strong Mutation Coverage	SMC	178
Weak Mutation Coverage	WMC	179

Bibliography

[1] W3C #28. Extensible markup language (XML) 1.0 (second edition) – W3C recommendation, October 2000. http://www.w3.org/XML/#9802xml10.

[2] Aynur Abdurazik and Jeff Offutt. Using UML collaboration diagrams for static checking and test generation. In *Third International Conference on the Unified Modeling Language (UML '00)*, pages 383–395, York, England, October 2000.

[3] Aynur Abdurazik and Jeff Offutt. Coupling-based class integration and test order. In *Workshop on Automation of Software Test (AST 2006)*, pages 50–56, Shanghai, China, May 2006.

[4] Alan T. Acree. *On Mutation*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, 1980.

[5] Alan T. Acree, Tim A. Budd, Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Mutation analysis. Technical report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, September 1979.

[6] S. B. Akers. On a theory of boolean functions. *Journal Society Industrial Applied Mathematics*, 7(4):487–498, December 1959.

[7] Roger T. Alexander. *Testing the Polymorphic Relationships of Object-oriented Programs*. PhD thesis, George Mason University, Fairfax, VA, 2001. Technical report ISE-TR-01-04, http://www.ise.gmu.edu/techrep/.

[8] Roger T. Alexander, James M. Bieman, Sudipto Chosh, and Bixia Ji. Mutation of Java objects. In *13th International Symposium on Software Reliability Engineering*, pages 341–351, Annapolis, MD, November 2002. IEEE Computer Society Press.

[9] Roger T. Alexander, James M. Bieman, and John Viega. Coping with Java programming stress. *IEEE Computer*, 33(4):30–38, 2000.

[10] Roger T. Alexander and Jeff Offutt. Analysis techniques for testing polymorphic relationships. In *Thirtieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '99)*, pages 104–114, Santa Barbara, CA, August 1999.

[11] Roger T. Alexander and Jeff Offutt. Criteria for testing polymorphic relationships. In *11th International Symposium on Software Reliability Engineering*, pages 15–23, San Jose, CA, October 2000. IEEE Computer Society Press.

[12] Roger T. Alexander and Jeff Offutt. Coupling-based testing of O-O programs. *Journal of Universal Computer Science*, 10(4):391–427, April 2004. http://www.jucs.org/jucs_10_4/coupling_based_testing_of.

[13] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–146, March 1976.

[14] Paul Ammann and Paul Black. A specification-based coverage metric to evaluate tests. In *4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 239–248, Washington, DC, November 1999.

[15] Paul Ammann, Paul Black, and W. Majurski. Using model checking to generate tests from specifications. In *2nd International Conference on Formal Engineering Methods*, pages 46–54, Brisbane, Australia, December 1998.

[16] Paul Ammann and Jeff Offutt. Using formal methods to derive test frames in category-partition testing. In *Ninth Annual Conference on Computer Assurance (COMPASS 94)*, pages 69–80, Gaithersburg MD, June 1994. IEEE Computer Society Press.

[17] Paul Ammann, Jeff Offutt, and Hong Huang. Coverage criteria for logical expressions. In *14th International Symposium on Software Reliability Engineering*, pages 99–107, Denver, CO, November 2003. IEEE Computer Society Press.

[18] Paul E. Ammann and John C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4):418–425, April 1988.

[19] D. M. St. Andre. Pilot mutation system (PIMS) user's manual. Technical report GIT-ICS-79/04, Georgia Institute of Technology, April 1979.

[20] Jo M. Atlee. Native model-checking of SCR requirements. In *Fourth International SCR Workshop*, November 1994.

[21] Jo M. Atlee and John Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[22] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[23] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Third Symposium on Software Testing, Analysis, and Verification*, pages 210–218, Key West Florida, December 1989. ACM SIGSOFT 89.

[24] Stephane Barbey and Alfred Strohmeier. The problematics of testing object-oriented software. In *SQM'94 Second Conference on Software Quality Management*, volume 2, pages 411–426, Edinburgh, Scotland, UK, 1994.

[25] Vic R. Basili and Richard W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, 13(12):1278–1296, December 1987.

[26] J. A. Bauer and A. B. Finger. Test plan generation using formal grammars. In *Fourth International Conference on Software Engineering*, pages 425–432, Munich, September 1979.

[27] Erich Beck and Kent Gamma. JUnit: A cook's tour. *Java Report*, 4(5):27–38, May 1999.

[28] Boris Beizer. *Software System Testing and Quality Assurance*. Van Nostrand, New York, 1984.

[29] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990. ISBN 0-442-20672-0.

[30] Michael Benedikt, Juliana Freire, and Patrice Godefroid. VeriWeb: Automatically testing dynamic Web sites. In *11th International World Wide Web Conference (WWW'2002) – Alternate Paper Tracks (WE-3 Web Testing and Maintenance)*, pages 654–668, Honolulu, HI, May 2002.

[31] Edward V. Berard. *Essays on Object-Oriented Software Engineering*, volume 1. Prentice Hall, New York, 1993.

[32] Philip J. Bernhard. A reduced test suite for protocol conformance testing. *ACM Transactions on Software Engineering and Methodology*, 3(3):201–220, July 1994.

[33] Robert Binder. *Testing Object-oriented Systems*. Addison-Wesley, New York, 2000.

[34] Robert V. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, September 1994.

[35] Robert V. Binder. Testing object-oriented software: A survey. *Software Testing, Verification and Reliability*, 6(3/4):125–252, 1996.

[36] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–345, 1983.

[37] Paul Black, Vladim Okun, and Y. Yesha. Mutation operators for specifications. In *Fifteenth IEEE International Conference on Automated Software Engineering*, pages 81–88, September 2000.

[38] Manuel Blum and Sampath Kannan. Designing programs that check their work. In *Twenty-First ACM Symposium on the Theory of Computing*, pages 86–97, 1989.

[39] Juris Borzovs, Audris Kalniņš, and Inga Medvedis. Automatic construction of test sets: Practical approach. In *Lecture Notes in Computer Science, Vol. 502*, pages 360–432. Springer Verlag, 1991.

[40] John H. Bowser. Reference manual for Ada mutant operators. Technical report GIT-SERC-88/02, Georgia Institute of Technology, February 1988.

[41] R. S. Boyer, B. Elpas, and K. N. Levitt. Select – a formal system for testing and debugging programs by symbolic execution. In *International Conference on Reliable Software*, June 1975. SIGPLAN Notices, vol. 10, no. 6.

[42] V. Braberman, M. Felder, and M. Marré. Testing timing behavior of real-time software. In *International Software Quality Week*, 1997.

[43] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C recommendation, February 1998. http://www.w3.org/TR/REC-xml/.

[44] Lionel Briand, J. Feng, and Yvan Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *14th International Conference on Software Engineering and Knowledge Engineering*, pages 43–50, Ischia, Italy, 2002. IEEE Computer Society Press.

[45] Lionel Briand and Yvan Labiche. A UML-based approach to system testing. In *Fourth International Conference on the Unified Modeling Language (UML '01)*, pages 194–208, Toronto, Canada, October 2001.

[46] Lionel Briand and Yvan Labiche. A UML-based approach to system testing. *Software and Systems Modeling*, 1(1):10–42, 2002.

[47] Lionel Briand, Yvan Labiche, and Yihong Wang. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. Technical report sce-01-02, Careleton University, 2001.

[48] Lionel Briand, Yvan Labiche, and Yihong Wang. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(7):594–607, July 2003.

[49] Lionel Briand, Massimiliano Di Penta, and Y. Labiche. Assessing and improving state-based class testing: A series of experiments. *IEEE Transactions on Software Engineering*, 30(11):770–793, November 2004.

[50] R. Brownlie, J. Prowse, and M. S. Phadke. Robust testing of AT&T PMX/StarMAIL using OATS. *AT&T Technical Journal*, 71(3):41–47, May/June 1992.

[51] Tim Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, November 1982.

[52] Tim Budd and Fred Sayward. Users guide to the Pilot mutation system. Technical report 114, Department of Computer Science, Yale University, 1977.

[53] Tim A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, 1980.

[54] Tim A. Budd, Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. The design of a prototype mutation system for program testing. In *NCC, AFIPS Conference Record*, pages 623–627, 1978.

[55] Tim A. Budd and Richard J. Lipton. Proving Lisp programs using test data. In *Digest for the Workshop on Software Testing and Test Documentation*, pages 374–403, Ft. Lauderdale, FL, December 1978. IEEE Computer Society Press.

[56] Tim A. Budd, Richard J. Lipton, Richard A. DeMillo, and Fred G. Sayward. Mutation analysis. Technical report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, April 1979.

[57] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *International Conference on Software Testing, Analysis, and Review (STAR'98)*, San Diego, CA, October 1998.

[58] K. Burroughs, A. Jain, and R. L. Erickson. Improved quality of protocol testing through techniques of experimental design. In *IEEE International Conference on Communications (Supercomm/ICC'94)*, pages 745–752, New Orleans, LA, May 1994. IEEE Computer Society Press.

[59] Ricky W. Butler and George B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *Software Engineering*, 19(1):3–12, 1993.

[60] Ugo Buy, Alessandro Orso, and Mauro Pezze. Automated testing of classes. In *2000 International Symposium on Software Testing, and Analysis (ISSTA '00)*, pages 39–48, Portland, OR, August 2000. IEEE Computer Society Press.

[61] R. Cardell-Oliver and T. Glover. A practical and complete algorithm for testing real-time systems. *Lecture Notes in Computer Science*, 1486:251–261, 1998.

[62] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Transactions on Software Engineering*, 5(4), July 1979.

[63] T. Y. Chen and M. F. Lau. Test case selection strategies based on boolean specifications. *Software Testing, Verification, and Reliability*, 11(3):165–180, June 2001.

[64] T. Y. Chen, P. L. Poon, S. F. Tang, and T. H. Tse. On the identification of categories and choices for specification-based test case generation. *Information and Software Technology*, 46(13):887–898, 2004.

[65] T. Y. Chen, S. F. Tang, P. L. Poon, and T. H. Tse. Identification of categories and choices in activity diagrams. In *Fifth International Conference on Quality Software (QSIC 2005)*, pages 55–63, Melbourne, Australia, September 2005. IEEE Computer Society Press.

[66] J. C. Cherniavsky. On finding test data sets for loop free programs. *Information Processing Letters*, 8(2):106–107, February 1979.

[67] J. C. Cherniavsky and C. H. Smith. A theory of program testing with applications. *Workshop on Software Testing*, pages 110–121, July 1986.

[68] Shing Chi Cheung, Samuel T. Chanson, and Zhendong Xu. Toward generic timing tests for distributed multimedia software systems. In *12th International Symposium on Software Reliability Engineering (ISSRE'01)*, page 210, Washington, DC, 2001. IEEE Computer Society Press.

[69] Philippe Chevalley. Applying mutation analysis for object-oriented programs using a reflective approach. In *8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, Macau SAR, China, December 2001.

[70] Philippe Chevalley and Pascale Thévenod-Fosse. Automated generation of statistical test cases from UML state diagrams. In *Proc. of IEEE 25th Annual International Computer Software and Applications Conference (COMPSAC2001)*, Chicago, IL, October 2001.

[71] Philippe Chevalley and Pascale Thévenod-Fosse. A mutation analysis tool for Java programs. *Journal on Software Tools for Technology Transfer (STTT)*, September 2002.

[72] John J. Chilenski. Personal communication, March 2003.

[73] John J. Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, September 1994.

[74] John J. Chilenski and L. A. Richey. Definition for a masking form of modified condition decision coverage (MCDC). Technical report, Boeing, Seattle, WA, 1997.

[75] Byoung-Ju Choi and Aditya Mathur. High-performance mutation testing. *Journal of Systems and Software*, 20(2):135–152, February 1993.

[76] N. Choquet. Test data generation using a prolog with constraints. In *Workshop on Software Testing*, pages 51–60, Banff, Alberta, July 1986. IEEE Computer Society Press.

[77] T. Chow. Testing software designs modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.

[78] D. Clarke and I. Lee. Automatic generation of tests for timing constraints from requirements. In *Third International Workshop on Object-Oriented Real-Time Dependable Systems*, Newport Beach, CA, February 1997.

[79] James M. Clarke. Automated test generation from a behavioral model. In *Software Quality Week Conference*, Brussels, Belgium, May 1998.

[80] Lori Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.

[81] Lori Clarke, Andy Podgurski, Debra Richardson, and Steven Zeil. A comparison of data flow path selection criteria. In *Eighth International Conference on Software Engineering*, pages 244–251, London, UK, August 1985. IEEE Computer Society Press.

[82] Lori Clarke, Andy Podgurski, Debra Richardson, and Steven Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15:1318–1332, November 1989.

[83] Lori Clarke and Debra Richardson. Applications of symbolic evaluation. *Journal of Systems and Software*, 5(1):15–35, January 1985.

[84] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, July 1997.

[85] David M. Cohen, Siddhartha R. Dalal, A. Kajla, and Gardner C. Patton. The automatic efficient test generator (AETG) system. In *Fifth International Symposium on Software Reliability Engineering (ISSRE'94)*, pages 303–309, Los Alamitos, CA, November 1994. IEEE Computer Society Press.

[86] David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, pages 83–88, September 1996.

[87] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colburn. Constructing test cases for interaction testing. In *25th International Conference on Software Engineering, (ICSE'03)*, pages 38–48. IEEE Computer Society, May 2003.

[88] L. L. Constantine and E. Yourdon. *Structured Design*. Prentice-Hall, Englewood Cliffs, NJ, 1979.

[89] Alan Cooper. *About Face: The Essentials of User Interface Design*. Hungry Minds, New York, 1995.

[90] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House Publishers, Norwood, MA, 2003.

[91] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, and C. M. Lott. Model-based testing of a highly programmable system. In *9th International Symposium in Software Engineering (ISSRE'98)*, pages 174–178, Paderborn, Germany, November 1998. IEEE Computer Society Press.

[92] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *21st International Conference on Software Engineering (ICSE'99)*, pages 285–294, Los Angeles, CA, May 1999. ACM Press.

[93] J. A. Darringer and J. C. King. Applications of symbolic execution to program testing. *IEEE Computer*, 11(4), April 1978.

[94] Márcio Delamaro, José C. Maldonado, and Aditya P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, March 2001.

[95] Márcio E. Delamaro and José C. Maldonado. Proteum – A tool for the assessment of test adequacy for C programs. In *Conference on Performability in Computing Systems (PCS 96)*, pages 79–95, New Brunswick, NJ, July 1996.

[96] Richard A. DeMillo, Dana S. Guindi, Kim N. King, W. Michael McCracken, and Jeff Offutt. An extended overview of the Mothra software testing environment. In *Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff, Alberta, July 1988. IEEE Computer Society Press.

[97] Richard A. DeMillo, Edward Krauser, and Aditya P. Mathur. Compiler-integrated program mutation. In *Fifteenth Annual Computer Software and Applications Conference (COMPSAC '92)*, Tokyo, Japan, September 1991. Kogakuin University, IEEE Computer Society Press.

[98] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5), May 1979.

[99] Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[100] Richard A. DeMillo, W. Michael McCracken, Rhonda J. Martin, and John F. Passafiume. *Software Testing and Evaluation*. Benjamin/Cummings, Menlo Park, CA, 1987.

[101] Richard A. DeMillo and Jeff Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[102] Richard A. DeMillo and Jeff Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering Methodology*, 2(2):109–127, April 1993.

[103] Dorothy Denning. *Cryptography and Data Security*. Addison Wesley, New York, 1982.

[104] M. S. Deutsch. *Software Verification and Validation Realistic Project Approaches*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[105] R. K. Doong and Phyllis G. Frankl. Case studies on testing object-oriented programs. In *Fourth Symposium on Software Testing, Analysis, and Verification*, pages 165–177, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.

[106] Mehmet Şahinoğlu and Eugene H. Spafford. A Bayes sequential statistical procedure for approving software products. In Wolfgang Ehrenberger, editor, *The IFIP Conference on Approving Software Products (ASP–90)*, pages 43–56, Garmisch-Partenkirchen, Germany, September 1990. Elsevier/North Holland, New York.

[107] A. G. Duncan and J. S. Hutchison. Using attributed grammars to test designs and implementations. In *Fifth International Conference on Software Engineering*, pages 170–178, San Diego, CA, March 1981.

[108] Arnaud Dupuy and Nancy Leveson. An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In *Digital Aviations Systems Conference (DASC)*, October 2000.

[109] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley Professional, New York, 1999.

[110] Dave E. Eckhardt Jr. and Larry D. Lee. Fundamental differences in the reliability of n-modular redundancy and n-version programming. *Journal of Systems and Software*, 8(4):313–318, September 1988.

[111] Alan Edelman. The mathematics of the Pentium division bug. *SIAM Review*, 39:54–67, March 1997. http://www.siam.org/journals/sirev/39-1/29395.html.

[112] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving Web application testing with user session data. In *25th International Conference on Software Engineering*, pages 49–59, Portland, OR, May 2003. IEEE Computer Society Press.

[113] Sebastian Elbaum, Gregg Rothermel, Srikanth Karre, and Marc Fisher. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, March 2005.

[114] R. En-Nouaary, Khendek F. Dssouli, and A. Elqortobi. Timed test case generation based on a state characterization technique. In *Proceeding of the 19th IEEE Real-Time Systems Symposium (RTSS98)*, Madrid, Spain, December 1998.

[115] Sadik Esmelioglu and Larry Apfelbaum. Automated test generation, execution, and reporting. In *Pacific Northwest Software Quality Conference*. IEEE Press, October 1997.

[116] R. E. Fairley. An experimental program testing facility. *IEEE Transactions on Software Engineering*, SE-1:350–3571, December 1975.

[117] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering Methodology*, 5(1):63–86, January 1996.

[118] S. P. Fiedler. Object-oriented unit testing. *Hewlett-Packard Journal*, 40(2):69–74, April 1989.

[119] Donald G. Firesmith. Testing object-oriented software. In *Testing Object-Oriented Languages and Systems (TOOLS)*, March 1993.

[120] Vladimir N. Fleyshgakker and Stewart N. Weiss. Efficient mutation analysis: A new approach. In *International Symposium on Software Testing and Analysis (ISSTA 94)*, pages 185–195, Seattle, WA, August 17–19 1994. ACM SIGSOFT, ACM Press.

[121] I. R. Forman. An algebra for data flow anomaly detection. In *Seventh International Conference on Software Engineering*, pages 278–286. Orlando, FL, March 1984. IEEE Computer Society Press.

[122] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305–330, September 1976.

[123] Phyllis G. Frankl and Yuetang Deng. Comparison of delivered reliability of branch, data flow and operational testing: A case study. In *2000 International Symposium on Software Testing, and Analysis (ISSTA '00)*, pages 124–134, Portland, OR, August 2000. IEEE Computer Society Press.

[124] Phyllis G. Frankl and Stewart N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[125] Phyllis G. Frankl, Stewart N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, 1997.

[126] Phyllis G. Frankl, Stewart N. Weiss, and Elaine J. Weyuker. ASSET: A system to select and evaluate tests. In *Conference on Software Tools*, New York, April 1985. IEEE Computer Society Press.

[127] Phyllis G. Frankl and Elaine J. Weyuker. Data flow testing in the presence of unexecutable paths. In *Workshop on Software Testing*, pages 4–13, Banff, Alberta, July 1986. IEEE Computer Society Press.

[128] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[129] Roy S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, June 1991.

[130] S. Fujiwara, G. Bochman, F. Khendek, M. Amalou, and A. Ghedasmi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[131] J. Gait. A probe effect in concurrent programs. *Software – Practice and Experience*, 16(3):225–233, March 1986.

[132] Leonard Gallagher, Jeff Offutt, and Tony Cincotta. Integration testing of object-oriented components using finite state machines. *Software Testing, Verification, and Reliability*, 17(1):215–266, January 2007.

[133] Robert Geist, Jeff Offutt, and Fred Harris. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions on Computers*, 41(5):550–558, May 1992. Special issue on Fault-Tolerant Computing.

[134] M. R. Girgis and M. R. Woodward. An integrated system for program testing using weak mutation and data flow analysis. In *Eighth International Conference on Software Engineering*, pages 313–319, London, UK, August 1985. IEEE Computer Society Press.

[135] M. R. Girgis and M. R. Woodward. An experimental comparison of the error exposing ability of program testing criteria. In *Workshop on Software Testing*, pages 64–73. Banff, Canada, July 1986. IEEE Computer Society Press.

[136] A. Goldberg, T. C. Wang, and D. Zimmerman. Applications of feasible path analysis to program testing. In *1994 International Symposium on Software Testing, and Analysis*, pages 80–94, Seattle, WA, August 1994.

[137] G. Gonenc. A method for the design of fault-detection experiments. *IEEE Transactions on Computers*, C-19:155–558, June 1970.

[138] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2), June 1975.

[139] J. S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, 9(6):686–709, November 1983.

[140] Mats Grindal. *Evaluation of Combination Strategies for Practical Testing*. PhD thesis, Skövde University / Linköping University, Skövde Sweden, 2007.

[141] Mats Grindal, Birgitta Lindström, Jeff Offutt, and Sten F. Andler. An evaluation of combination testing strategies. *Empirical Software Engineering*, 11(4):583–611, December 2006.

[142] Mats Grindal and Jeff Offutt. Input parameter modeling for combination strategies. In *IASTED International Conference on Software Engineering (SE 2007)*, Innsbruck, Austria, February 2007. ACTA Press.

[143] Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: A survey. *Software Testing, Verification, and Reliability*, 15(2):97–133, September 2005.

[144] Mats Grindal, Jeff Offutt, and Jonas Mellin. Conflict management when using combination strategies for software testing. In *Australian Software Engineering Conference (ASWEC 2007)*, pages 255–264, Melbourne, Australia, April 2007.

[145] Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Software Testing, Verification, and Reliability*, 3(2):63–82, 1993.

[146] Matthias Grochtmann, Klaus Grimm, and J. Wegener. Tool-supported test case design for black-box testing by means of the classification-tree editor. In *1st European International Conference on Software Testing Analysis & Review (EuroSTAR 1993)*, pages 169–176, London, UK, October 1993.

[147] Richard Hamlet. Reliability theory of program testing. *Acta Informatica*, pages 31–43, 1981.

[148] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[149] Richard G. Hamlet and Richard Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

[150] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 4:242–257, 1970.

[151] Jim M. Hanks. Testing cobol programs by mutation: Volume I – Introduction to the CMS.1 system, Volume II – CMS.1 system documentation. Technical report GIT-ICS-80/04, Georgia Institute of Technology, February 1980.

[152] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. In *Symposium on Foundations of Software Engineering*, pages 154–163, New Orleans, LA, December 1994. ACM SIGSOFT.

[153] Mary Jean Harrold and Gregg Rothermel. Empirical studies of a safe regression test selection technque. *IEEE Transactions on Software Engineering*, 24(6):401–419, June 1998.

[154] Mary Jean Harrold and Mary Lou Soffa. Selecting and using data for integration testing. *IEEE Software*, 8(2):58–65, March 1991.

[155] Mats E. Heimdahl. Safety and software intensive systems: Challenges old and new. In *International Conference on Software Engineering: Future of Software Engineering*, pages 137–152, May 2007.

[156] E. Heller. Using design of experiment structures to generate software test cases. In *12th International Conference on Testing Computer Software*, pages 33–41, New York, 1995. ACM.

[157] K. Henninger. Specifiying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, SE-6(1):2–12, January 1980.

[158] P. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3):92–96, November 1976.

[159] A. Hessel, K. Larsen, B. Nielsen, and A Skou. Time optimal real-time test case generation using UPPAAL. In *Workshop on Formal Approaches to Testing of Software (FATES)*, Montreal, October 2003.

[160] Bill Hetzel. *The Complete Guide to Software Testing*. Wiley-QED, second edition, 1988.

[161] J. Robert Horgan and Saul London. Data flow coverage and the C language. In *Fourth Symposium on Software Testing, Analysis, and Verification*, pages 87–97, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.

[162] J. Robert Horgan and Saul London. ATAC: A data flow coverage testing tool for C. In *Symposium of Quality Software Development Tools*, pages 2–10, New Orleans, LA, May 1992. IEEE Computer Society Press.

[163] J. Robert Horgan and Aditya P. Mathur. Weak mutation is probably strong mutation. Technical report SERC-TR-83-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, December 1990.

[164] Willam E. Howden. Methodology for the generation of program test data. *IEEE Transactions on Software Engineering*, SE-24, May 1975.

[165] Willam E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, September 1976.

[166] Willam E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4), July 1977.

[167] Willam E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[168] Willam E. Howden. The theory and practice of function testing. *IEEE Software*, 2(5), September 1985.

[169] William E. Howden. *Functional Program Testing and Analysis*. McGraw-Hill, New York, 1987.

[170] J. C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3):113–128, September 1975.

[171] J. Huller. Reducing time to market with combinatorial design method testing. In *10th Annual International Council on Systems Engineering (INCOSE'00)*, Minneapolis, MN, July 2000.

[172] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Sixteenth International Conference on Software Engineering*, pages 191–200, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[173] Michael Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, Cambridge, UK, 2000.

[174] G. Hwang, K. Tai, and T. Hunag. Reachability testing: An approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering*, 5(4), December 1995.

[175] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. ANSI/IEEE Std 610.12-1990, 1996.

[176] D. C. Ince. The automatic generation of test data. *Computer Journal*, 30(1):63–69, 1987.

[177] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and feasible path analysis. In *1994 International Symposium on Software Testing, and Analysis*, pages 95–107, Seattle, WA, August 1994.

[178] Zhenyi Jin and Jeff Offutt. Coupling-based criteria for integration testing. *Software Testing, Verification, and Reliability*, 8(3):133–154, September 1998.

[179] B. F. Jones, D. E. Eyres, and H. H. Sthamer. A strategy for using genetic algorithms to automate branch and fault-based testing. *Computer Journal*, 41(2):98–107, 1998.

[180] J. A. Jones and Mary Jean Harrold. Test-suite reduction and prioritizaion for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, March 2003.

[181] Paul C. Jorgensen and Carl Erickson. Object-oriented integration testing. *Communications of the ACM*, 37(9):30–38, 1994.

[182] Cem Kaner, Jack Falk, and Hung Q. Nguyen. *Testing Computer Software*. John Wiley and Sons, New York NY, second edition, 1999.

[183] David J. Kasik and Harry G. George. Toward automatic generation of novice user test scripts. In *Conference on Human Factors in Computing Systems: Common Ground*, pages 244–251, New York, April 1996.

[184] Sun-Woo Kim, John Clark, and John McDermid. Assessing test set adequacy for object-oriented programs using class mutation. In *of Symposium on Software Technology (SoST'99)*, pages 72–83, September 1999.

[185] Sunwoo Kim, John A. Clark, and John A. McDermid. Investigating the effectiveness of object-oriented strategies with the mutation method. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 4–100, San Jose, CA, October 2000. Special issue of the *Journal of Software Testing, Verification, and Reliability*, December 2001.

[186] Y. G. Kim, H. S. Hong, S. M. Cho, D. H. Bae, and S. D. Cha. Test cases generation from UML state diagrams. *IEE Proceedings – Software*, 146(4):187–192, August 1999.

[187] Kim N. King and Jeff Offutt. A Fortran language system for mutation-based software testing. *Software – Practice and Experience*, 21(7):685–718, July 1991.

[188] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[189] Charles Knutson and Sam Carmichael. Safety first: Avoiding software mishaps, November 2000. http://www.embedded.com/2000/0011/0011feat1.htm.

[190] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[191] Bogdan Korel. Dynamic method for software test data generation. *Software Testing, Verification, and Reliability*, 2(4):203–213, 1992.

[192] Edward W. Krauser, Aditya P. Mathur, and Vernon Rego. High performance testing on SIMD machines. In *Second Workshop on Software Testing, Verification, and Analysis*, pages 171–177, Banff, Alberta, July 1988. IEEE Computer Society Press.

[193] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *The SPIN'04 Workshop on Model-Checking Software*, Barcelona, Spain, 2004.

[194] D. R. Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering Methodology*, 8(4):411–424, October 1999.

[195] D. R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In *27th NASA/IEE Software Engineering Workshop*, NASA Goodard Space Flight Center, MD, December 2002. NASA/IEEE.

[196] D. R. Kuhn, D. R. Wallace, and A. M. Gallo Jr. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, June 2004.

[197] D. Kung, J. Gao, Pei Hsia, Y. Toyoshima, and C. Chen. A test strategy for object-oriented programs. In *19th Computer Software and Applications Conference (COMPSAC '95)*, pages 239–244, Dallas, TX, August 1995. IEEE Computer Society Press.

[198] D. Kung, C. H. Liu, and P. Hsia. An object-oriented Web test model for testing Web applications. In *IEEE 24th Annual International Computer Software and Applications Conference (COMPSAC2000)*, pages 537–542, Taipei, Taiwan, October 2000.

[199] W. LaLonde and J. Pugh. Subclassing != subtyping != is-a. *Journal of Object Oriented Programming*, 3(5): 57–62, January 1991.

[200] Janusz Laski. Data flow testing in STAD. *Journal of Systems and Software*, 12:3–14, 1990.

[201] Janusz Laski and Bogdan Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9(3):347–354, May 1983.

[202] M. F. Lau and Y. T. Yu. An extended fault class hierarchy for specification-based testing. *ACM Transactions on Software Engineering Methodology*, 14(3):247–276, July 2005.

[203] Suet Chun Lee and Jeff Offutt. Generating test cases for XML-based web component interactions using mutation analysis. In *12th International Symposium on Software Reliability Engineering*, pages 200–209, Hong Kong, China, November 2001. IEEE Computer Society Press.

[204] H. Legard and M. Marcotty. A generalogy of control structures. *Communications of the ACM*, 18:629–639, November 1975.

[205] Yu Lei and K. C. Tai. In-parameter-order: A test generation strategy for pairwise testing. In *Third IEEE High Assurance Systems Engineering Symposium*, pages 254–261, November 1998. IEEE Computer Society Press.

[206] Yu Lei and K. C. Tai. A test generation strategy for pairwise testing. Technical Report TR-2001-03, Department of Computer Science, North Carolina State University, Raleigh, 2001.

[207] Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, Reading, MA, 1995.

[208] Zhang Li, Mark Harman, and Rob M. Hierons. Meta-heuristic search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, April 2007.

[209] J. L. Lions. Ariane 5 flight 501 failure: Report by the inquiry board, July 1996. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

[210] Richard Lipton. New directions in testing. In *Distributed Computing and Cryptography, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 2, pages 191–202, Providence, RI, 1991.

[211] Richard J. Lipton and Fred G. Sayward. The status of research on program mutation. In *Digest for the Workshop on Software Testing and Test Documentation*, pages 355–373, December 1978.

[212] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison Wesley, New York, 2000.

[213] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(1):1811–1841, November 1994.

[214] B. Littlewood and D. R. Miller. Conceptual modeling of coincident failures in multiversion software. *IEEE Transactions on Software Engineering*, 15(12):1596–1614, December 1989.

[215] C. H. Liu, D. Kung, P. Hsia, and C. T. Hsu. Structural testing of Web applications. In *11th International Symposium on Software Reliability Engineering*, pages 84–96, San Jose, CA, October 2000. IEEE Computer Society Press.

[216] C. H. Liu, D. C. Kung, P. Hsia, and C. T. Hsu. An object-based data flow testing approach for Web applications. *International Journal of Software Engineering and Knowledge Engineering*, 11(2):157–179, 2001.

[217] Giuseppe A. Di Lucca and Massimiliano Di Penta. Considering browser interaction in web application testing. In *5th International Workshop on Web Site Evolution (WSE 2003)*, pages 74–84, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

[218] Stephen F. Lundstrom. Adaptive random data generation for computer software testing. In *National Computer Conference*, pages 505–511, 1978.

[219] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for Java. In *13th International Symposium on Software Reliability Engineering*, pages 352–363, Annapolis, MD, November 2002. IEEE Computer Society Press.

[220] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava : An automated class mutation system. *Software Testing, Verification, and Reliability*, 15(2):97–133, June 2005.

[221] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. mujava home page. online, 2005. http://ise.gmu.edu/ offutt/mujava/, http://salmosa.kaist.ac.kr/LAB/MuJava/, last access April 2006.

[222] Yashwant K. Malaiya. Antirandom testing: Getting the most out of black-box testing. In *International Symposium on Software Reliability Engineering (ISSRE'95)*, pages 86–95, Toulouse, France, October 1995.

[223] Brian A. Malloy, Peter J. Clarke, and Errol L. Lloyd. A parameterized cost model to order classes for class-based testing of C++ applications. In *14th International Symposium on Software Reliability Engineering*, Denver, CO, 2003. IEEE Computer Society Press.

[224] Robert Mandl. Orthogonal latin squares: An application of experiment design to compiler testing. *Communications of the ACM*, 28(10):1054–1058, October 1985.

[225] D. Mandrioli, S. Morasca, and A. Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Transactions on Computer Systems*, 4(13):365–398, Nov. 1995.

[226] Brian Marick. The weak mutation hypothesis. In *Fourth Symposium on Software Testing, Analysis, and Verification*, pages 190–199, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.

[227] Brian Marick. *The Craft of Software Testing: Subsystem Testing, Including Object-Based and Object-Oriented Testing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[228] Aditya P. Mathur. On the relative strengths of data flow and mutation based test adequacy criteria. In *Sixth Annual Pacific Northwest Software Quality Conference*, Portland, OR, Lawrence and Craig, 1991.

[229] Aditya P. Mathur and Edward W. Krauser. Mutant unification for improved vectorization. Technical report SERC-TR-14-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, April 1988.

[230] Aditya P. Mathur and W. Eric Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification, and Reliability*, 4(1):9–31, March 1994.

[231] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, July 1990.

[232] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[233] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley, New York, 2006.

[234] Phil McMinn. Search-based software test data generation: A survey. *Software Testing, Verification, and Reliability*, 13(2):105–156, June 2004.

[235] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Using a goal-driven approach to generate test cases for GUIs. In *21st International Conference on Software Engineering*, pages 257–266, May 1999.

[236] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Automated test oracles for GUIs. In *ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-8)*, pages 30–39, New York, November 2000.

[237] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, February 2001.

[238] Atif M. Memon and Mary Lou Soffa. Regression testing of GUIs. In *9th European Software Engineering Conference (ESEC) and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11)*, pages 118–127, September 2003.

[239] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for GUI testing. In *8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, pages 256–267, September 2001.

[240] Atif M. Memon and Qing Xie. Empirical evaluation of the fault-detection effectiveness of smoke regression test cases for GUI-based software. In *The International Conference on Software Maintenance 2004 (ICSM'04)*, pages 8–17, Washington, DC, September 2004.

[241] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, second edition, 1997.

[242] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, December 2001.

[243] E. F. Miller and R. A. Melton. Automated generation of testcase datasets. In *International Conference on Reliable Software*, pages 51–58, April 1975.

[244] Cleve Moler. A tale of two numbers. *SIAM News*, 28(1), January 1995.

[245] S. Morasca and Mauro Pezze. Using high level Petri-nets for testing concurrent and real-time systems. *Real-Time Systems: Theory and Applications*, pages 119–131. Amsterdam, North-Holland, 1990.

[246] Larry J. Morell. *A Theory of Error-Based Testing*. PhD thesis, University of Maryland, College Park, MD, 1984. Technical Report TR-1395.

[247] Larry J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[248] Carlos Urias Munoz. An approach to software product testing. *IEEE Transactions on Software Engineering*, 14(11):1589–1595, November 1988.

[249] Glenford Myers. *The Art of Software Testing*. John Wiley and Sons, New York, 1979.

[250] Adithya Nagarajan and Atif M. Memon. Refactoring using event-based profiling. In *First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*, Victoria, British Columbia, November 2003.

[251] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition tours. In *Fault Tolerant Computing Systems*, pages 238–243. IEEE Computer Society Press, 1981.

[252] B. Nielsen and A. Skou. Automated test generation from timed automata. In *21st IEEE Real-Time Systems Symposium*, Walt Disney World, Orlando, FL, 2000. IEEE Computer Society Press.

[253] Jacok Nielsen. *Designing Web Usability*. New Riders Publishing, Indianapolis, IN, 2000.

[254] Robert Nilsson. *Automatic Timeliness Testing of Dynamic Real-Time Systems*. PhD thesis, Skövde University/Linköping University, Skövde Sweden, 2006.

[255] Robert Nilsson and Jeff Offutt. Automated testing of timeliness : A case study. In *Second Workshop on Automation of Software Test (AST 2007)*, Minneapolis, MN, May 2007.

[256] Robert Nilsson, Jeff Offutt, and Sten F. Andler. Mutation-based testing criteria for timeliness. In *28th Annual International Computer Software and Applications Conference (COMPSAC 2004)*, pages 306–312, Hong Kong, September 2004.

[257] Robert Nilsson, Jeff Offutt, and Jonas Mellin. Test case generation for mutation-based testing of timeliness. In *2nd International Workshop on Model Based Testing*, pages 102–121, Vienna, Austria, March 2006.

[258] S. C. Ntafos. An evaluation of required element testing strategies. In *Seventh International Conference on Software Engineering*, pages 250–256, Orlando FL, March 1984. IEEE Computer Society Press.

[259] Bashar Nuseibeh. Who dunnit? *IEEE Software*, 14:15–16, May/June 1997.

[260] Department of Defense. *DOD-STD-2167A: Defense System Software Development*. Department of Defense, February 1988.

[261] Department of Defense. *MIL-STD-498: Software Development and Documentation*. Department of Defense, December 1994.

[262] Jeff Offutt. *Automatic Test Data Generation*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, 1988. Technical report GIT-ICS 88/28.

[263] Jeff Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.

[264] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Second IEEE International Conference on the Unified Modeling Language (UML99)*, pages 416–429, Fort Collins, CO, October 1999. Springer-Verlag Lecture Notes in Computer Science Volume 1723.

[265] Jeff Offutt, Aynur Abdurazik, and Roger T. Alexander. An analysis tool for coupling-based integration testing. In *The Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00)*, pages 172–178, Tokyo, Japan, September 2000. IEEE Computer Society Press.

[266] Jeff Offutt, Roger Alexander, Ye Wu, Quansheng Xiao, and Chuck Hutchinson. A fault model for subtype inheritance and polymorphism. In *12th International Symposium on Software Reliability Engineering*, pages 84–93, Hong Kong, China, November 2001. IEEE Computer Society Press.

[267] Jeff Offutt, Zhenyi Jin, and Jie Pan. The dynamic domain reduction approach to test data generation. *Software – Practice and Experience*, 29(2):167–193, January 1999.

[268] Jeff Offutt and Kim N. King. A Fortran 77 interpreter for mutation analysis. In *1987 Symposium on Interpreters and Interpretive Techniques*, pages 177–188, St. Paul MN, June 1987. ACM SIGPLAN.

[269] Jeff Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.

[270] Jeff Offutt and Stephen D. Lee. How strong is weak mutation? In *Fourth Symposium on Software Testing, Analysis, and Verification*, pages 200–213, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.

[271] Jeff Offutt and Stephen D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, May 1994.

[272] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification, and Reliability*, 13(1):25–53, March 2003.

[273] Jeff Offutt and Jie Pan. Detecting equivalent mutants and the feasible path problem. *Software Testing, Verification, and Reliability*, 7(3):165–192, September 1997.

[274] Jeff Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. An experimental evaluation of data flow and mutation testing. *Software – Practice and Experience*, 26(2):165–176, February 1996.

[275] Jeff Offutt, Roy Pargas, Scott V. Fichter, and P. Khambekar. Mutation testing of software using a MIMD computer. In *1992 International Conference on Parallel Processing*, pages II257–266, Chicago, August 1992.

[276] Jeff Offutt, Jeffrey Payne, and Jeffrey M. Voas. Mutation operators for Ada. Technical report ISSE-TR-96-09, Department of Information and Software Systems Engineering, George Mason University, Fairfax, VA, March 1996. http://www.ise.gmu.edu/techrep/.

[277] Jeff Offutt and Roland Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, San Jose, CA, October 2000.

[278] Jeff Offutt, Ye Wu, Xiaochen Du, and Hong Huang. Bypass testing of web applications. In *15th International Symposium on Software Reliability Engineering*, pages 187–197, Saint-Malo, Bretagne, France, November 2004. IEEE Computer Society Press.

[279] Jeff Offutt and Wuzhi Xu. Generating test cases for web services using data perturbation. In *Workshop on Testing, Analysis and Verification of Web Services*, Boston, MA, July 2004. ACM SIGSoft.

[280] Jeff Offutt and Wuzhi Xu. Testing web services by XML perturbation. In *16th International Symposium on Software Reliability Engineering*, Chicago, IL, November 2005. IEEE Computer Society Press.

[281] Alex Orso and Mauro Pezze. Integration testing of procedural object oriented programs with polymorphism. In *Sixteenth International Conference on Testing Computer Software*, pages 103–114, Washington, DC, June 1999. ACM SIGSOFT.

[282] L. J. Osterweil and L. D. Fosdick. Data flow analysis as an aid in documentation, assertion generation, validation, and error detection. Technical report CU-CS-055-74, Department of Computer Science, University of Colorado, Boulder, CO, September 1974.

[283] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[284] Jan Overbeck. *Integration Testing for Object-Oriented Software*. PhD dissertation, Vienna University of Technology, 1994.

[285] A. J. Payne. A formalised technique for expressing compiler exercisers. *SIGLPAN Notices*, 13(1):59–69, January 1978.

[286] Ivars Peterson. Pentium bug revisited, May 1997. http://www.maa.org/mathland/mathland_5_12.html.

[287] E. Petitjean and H. Fochal. A realistic architecture for timed testing. In *Fifth IEEE International Conference on Engineering of Complex Computer Systems*, Las Vegas, October 1999.

[288] A. Pettersson and H. Thane. Testing of multi-tasking real-time systems with critical sections. In *Proceedings of Ninth International Conference on Real-Time Computing Systems and Applications (RTCSA'03)*, Tainan City, Taiwan, February 2003.

[289] Mauro Pezze and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley, Hoboken, NJ, 2008.

[290] S. Pimont and J. C. Rault. A software reliability assessment based on a structural behavioral analysis of programs. In *Second International Conference on Software Engineering*, San Francisco, CA, October 1976.

[291] P. Piwowarski, M. Ohba, and J. Caruso. Coverage measure experience during function test. In *14th International Conference on Software Engineering (ICSE'93)*, pages 287–301, Los Alamitos, CA, May 1993. ACM.

[292] T. A. Powell. *Web Site Engineering: Beyond Web Page Design*. Prentice-Hall, Englewood Cliffs, NJ, 2000.

[293] R. E. Prather. Theory of program testing – an overview. *Bell System Technical Journal*, 62(10):3073–3105, December 1983.

[294] Paul Purdom. A sentence generator for testing parsers. *BIT*, 12:366–375, 1972.

[295] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300, December 1976.

[296] K. Ramamritham. The origin of time constraints. In *First International Workshop on Active and Real-Time Database Systems (ARTDB 1995)*, pages 50–62, Skövde, Sweden, June 1995. Springer, New York, 1995.

[297] S. Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[298] S. Rayadurgam and M. P. E. Heimdahl. Coverage based test-case generation using model checkers. In *8th IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 83–91, April 2001.

[299] Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs, and Daniel Weber. Automatic testing of reactive systems. In *Proceeding of the 19th IEEE Real-Time Systems Symposium (RTSS98)*, 1998.

[300] F. Ricca and P. Tonella. Analysis and testing of Web applications. In *23rd International Conference on Software Engineering (ICSE '01)*, pages 25–34, Toronto, CA, May 2001.

[301] Marc Roper. *Software Testing*. International Software Quality Assurance Series. McGraw-Hill, New York, 1994.

[302] Dave Rosenblum and Gregg Rothermel. A comparative study of regression test selection techniques. In *IEEE Computer Society 2nd International Workshop on*

Empirical Studies of Software Maintenance, pages 89–94, Bari, Italy, October 1997. IEEE Computer Society Press.

[303] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[304] J. Rowland and Y. Zuyuan. Experimental comparison of three system test strategies preliminary report. In *Third Symposium on Software Testing, Analysis, and Verification*, pages 141–149, Key West, FL, December 1989. ACM SIGSOFT 89.

[305] RTCA-DO-178B. Software considerations in airborne systems and equipment certification, December 1992.

[306] Avi D. Rubin. *White-Hat Secuirty Arsenal: Tackling the Threats*. Addison-Wesley, New York, 2001.

[307] K. Sabnani and A. Dahbura. A protocol testing procedure. *Computer Networks and ISDN Systems*, 14(4):285–297, 1988.

[308] P. SanPietro, A. Morzenti, and S. Morasca. Generation of execution sequences for modular time critical systems. *IEEE Transactions on Software Engineering*, 26(2):128–149, February 2000.

[309] W. Schütz. *The Testability of Distributed Real-Time Systems*. Kluwer Academic Publishers, Norwell, MA, 1993.

[310] W. Schütz. Fundamental issues in testing distributed real-time systems. *Real-Time Systems*, 7(2):129–157, September 1994.

[311] Richard W. Selby. Combining software testing strategies: An empirical evaluation. In *Workshop on Software Testing*, pages 82–90, Banff, Alberta, July 1986. IEEE Computer Society Press.

[312] Richard K. Shehady and Daniel P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *27th International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 80–88, Washington, Brussels, Tokyo, June 1997.

[313] G. Sherwood. Effective testing of factor combinations. In *Third International Conference on Software Testing, Analysis, and Review (STAR94)*, Washington, DC, May 1994. Software Quality Engineering.

[314] T. Shiba, T. Tsuchiya, and T. Kikuno. Using artificial life techniques to generate test cases for combinatorial testing. In *28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, pages 72–77, Hong Kong, China, September 2004. IEEE Computer Society Press.

[315] M. D. Smith and D. J. Robson. Object-oriented programming: The problems of validation. In *6th International Conference on Software Maintenance*, pages 272–282, Los Alamitos, CA, 1990. IEEE Computer Society Press.

[316] Ian Sommerville. *Software Engineering*. Addison-Wesley, New York, 6th edition, 2001.

[317] British Computer Society Specialist Interest Group in Software Testing. *Standard for Software Component Testing, Working Draft 3.3*. British Computer Society, 1997. http://www.rmcs.cranfield.ac.uk/~cised/sreid/BCS_SIG/.

[318] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[319] Phil Stocks and Dave Carrington. Test templates: A specification-based testing framework. In *Fifteenth International Conference on Software Engineering*, pages 405–414, Baltimore, MD, May 1993.

[320] Phil Stocks and Dave Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[321] K. C. Tai and F. J. Daniels. Test order for inter-class integration testing of object-oriented software. In *The Twenty-First Annual International Computer Software and Applications Conference (COMPSAC '97)*, pages 602–607, Santa Barbara, CA, 1997. IEEE Computer Society.

[322] K. C. Tai and Yu Lei. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28(1):109–111, January 2002.

[323] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.

[324] M. Tatsubori. OpenJava WWW page. Tokyo Institute of Technology, Chiba Shigeru Group, 2002. http://www.csg.is.titech.ac.jp/~mich/openjava/ (accessed May 2004).

[325] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. *Reflection and Software Engineering*, LNCS 1826:117–133, June 2000.

[326] H. Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Royal Institute of Technology, KTH, Stockholm, Sweden, 2000.

[327] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. An experimental study on software structural testing: Deterministic versus random input generation. In *Fault-Tolerant Computing: The Twenty-First International Symposium*, pages 410–417, Montreal, Canada, June 1991. IEEE Computer Society Press.

[328] F. Tip. A survey of program slicing techniques. Technical report CS-R-9438, Computer Science/Department of Software Technology, Centrum voor Wiskunde en Informatica, 1994.

[329] Yves Le Traon, Thierry Jéron, Jean-Marc Jézéquel, and Pierre Morel. Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability*, 49(1):12–25, March 2000.

[330] Roland H. Untch, Jeff Offutt, and Mary Jean Harrold. Mutation analysis using program schemata. In *1993 International Symposium on Software Testing, and Analysis*, pages 139–148, Cambridge, MA, June 1993.

[331] P. Verissimo and H. Kopetz. Design of distributed real-time systems. In S. Mullender, editor, *Distributed Systems*, pages 511–530. Addison-Wesley, New York, 1993.

[332] S. A. Vilkomir and J. P. Bowen. Reinforced condition/decision coverage (RC/DC): A new criterion for software testing. In *ZB2002: 2nd International Conference of Z and B Users*, pages 295–313, Grenoble, France, January 2002. Springer-Verlag, LNCS 2272.

[333] Jeffrey M. Voas. Pie: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, August 1992.

[334] Jeffrey M. Voas and Keith W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):553–563, May 1995.

[335] K. S. How Tai Wah. Fault coupling in finite bijective functions. *Software Testing, Verification, and Reliability*, 5(1):3–47, March 1995.

[336] K. S. How Tai Wah. A theoretical study of fault coupling. *Software Testing, Verification, and Reliability*, 10(1):3–46, March 2000.

[337] A. Watkins, D. Berndt, K. Aebischer, J. Fisher, and L. Johnson. Breeding software test cases for complex systems. In *37th Annual Hawaii International Conference on System Sciences (HICSS'04) – Track 9*, page 90303.3, Washington, DC, USA, 2004. IEEE Computer Society Press.

[338] J. Wegener, H. H. Sthammer, B. F. Jones, and D. E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997.

[339] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[340] Steward N. Weiss. What to compare when comparing test data adequacy criteria. *ACM SIGSOFT Notes*, 14(6):42–49, October 1989.

[341] Elaine Weyuker. The oracle assumption of program testing. In *Thirteenth International Conference on System Sciences*, pages 44–49, Honolulu, HI, January 1980.

[342] Elaine Weyuker, Thomas Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.

[343] Elaine J. Weyuker and Thomas J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3):236–246, May 1980.

[344] Elaine J. Weyuker, Stewart N. Weiss, and Richard G. Hamlet. Comparison of program testing strategies. In *Fourth Symposium on Software Testing, Analysis, and Verification*, pages 1–10, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.

[345] Elaine J. Weyuker, Stewart N. Weiss, and Richard G. Hamlet. Data flow-based adequacy analysis for languages with pointers. In *Fourth Symposium on Software Testing, Analysis, and Verification*, pages 74–86, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.

[346] Lee White and Husain Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *9th International Symposium on Software Reliability Engineering*, pages 110–121, October 2000.

[347] Lee White, Husain Almezen, and Nasser Alzeidi. User-based testing of GUI sequences and their interaction. In *10th International Symposium on Software Reliability Engineering*, pages 54–63, November 2001.

[348] Lee White and Bogdan Wiszniewski. Path testing of computer programs with loops using a tool for simple loop patterns. *Software – Practice and Experience*, 21(10):1075–1102, October 1991.

[349] Lee J. White. Software testing and verification. In Marshall C. Yovits, editor, *Advances in Computers*, volume 26, pages 335–390. Academic Press, New York, 1987.

[350] Duminda Wijesekera, Lingya Sun, Paul Ammann, and Gordon Fraser. Relating counterexamples to test cases in CTL model checking specifications. In *A-MOST*

'07: Third ACM Workshop on the Advances in Model-Based Testing, co-located with ISSTA 2007, London, UK, July 2007.

[351] Christian Wild, Steven Zeil, and Gao Feng. Employing accumulated knowledge to refine test descriptions. *Software Testing, Verification, and Reliability*, July 2(2): 53–68, 1992.

[352] Alan W. Williams. Determination of test configurations for pair-wise interaction coverage. In *13th International Conference on the Testing of Communicating Systems (TestCom 2000)*, pages 59–74, Ottawa, Canada, August 2000.

[353] Alan W. Williams and Robert L. Probert. A practical strategy for testing pair-wise coverage of network interfaces. In *7th International Symposium on Software Reliability Engineering (ISSRE96)*, White Plains, New York, November 1996.

[354] Alan W. Williams and Robert L. Probert. A measure for component interaction test coverage. In *ACSI/IEEE International Conference on Computer Systems and Applications (AICCSA 2001)*, pages 304–311, Beirut, Lebanon, June 2001.

[355] Barbara Liskov with John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, New York, 2001.

[356] W. Eric Wong. *On Mutation and Data Flow*. PhD thesis, Purdue University, December 1993. (Also Technical Report SERC-TR-149-P, Software Engineering Research Center, Purdue University, West Lafayette, IN).

[357] W. Eric Wong and Aditya P. Mathur. Fault detection effectiveness of mutation and data flow testing. *Software Quality Journal*, 4(1):69–83, March 1995.

[358] M. R. Woodward and K. Halewood. From weak to strong, dead or alive? An analysis of some mutation testing issues. In *Second Workshop on Software Testing, Verification, and Analysis*, pages 152–158, Banff Alberta, July 1988. IEEE Computer Society Press.

[359] Ye Wu and Jeff Offutt. Modeling and testing Web-based applications. Technical report ISE-TR-02-08, Department of Information and Software Engineering, George Mason University, Fairfax, VA, July 2002. http://www.ise.gmu.edu/techrep/.

[360] Ye Wu, Jeff Offutt, and Xiaochen Du. Modeling and testing of dynamic aspects of Web applications. Technical report ISE-TR-04-01, Department of Information and Software Engineering, George Mason University, Fairfax, VA, July 2004. http://www.ise.gmu.edu/techrep/.

[361] Tao Xie and Dave Notkin. Checking inside the black box: Regression testing by comparing value spectra. *IEEE Transactions on Software Engineering*, 31(10):869–883, October 2005.

[362] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 45–54, Boston, MA, July 2004. ACM Software Engineering Notes.

[363] H. Yin, Z. Lebne-Dengel, and Y. K. Malaiya. Automatic test generation using checkpoint encoding and antirandom testing. Technical Report CS-97-116, Colorado State University, 1997.

[364] S. J. Young. *Real-Time Languages: Design and Development*. Ellis Horwood, Chichester, UK, 1982.

[365] Christian N. Zapf. Medusamothra – A distributed interpreter for the mothra mutation testing system. M.S. thesis, Clemson University, Clemson, SC, August 1993.

[366] Hong Zhu. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Transactions on Software Engineering*, 22(4):248–255, April 1996.

[367] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

Index

active clause coverage (ACC)
  ambiguity, 108–109
  definition, 108
activity diagram , 89–90
All-Coupling-Defs-Uses (ACDU), 255
All-Coupling-Sequences (ACS), 255
All-Poly-Classes (APC), 255
All-Poly-Coupling-Defs-Uses (APCDU), 256
ANSI/IEEE Standard, 225
architectural design, 6
Ariane rocket, 8
automatic test data generation, 289–290

basic block, 28, 52–55, 178, 186, 259, 270, 278
  definition, 52
best effort touring, *see* tour, best effort
black-box testing
  definition, 21
block, 150, 153
bottom-up testing
  definition, 22
bypass testing, 258

characteristic, 151, 153
  examples, 151, 152, 155
  functionality-based, 153–155, 158
  interface-based, 153–157
CITO, *see* class integration test order
class integration test order, 218–219, 222, 237
  definition, 219
clause
  definition, 105
combination strategy, 160
component, 6, 11, 65, 73, 191–192, 217–218, 236–237, 256–260
  definition, 217
concurrency, 268
connector, 6
control flow graph, 259
controllability, 120, 263, 284–286
  definition, 14
coupling path, 247–256
coupling sequence, 250–256
coverage analysis tool, 269
  definition, 268
coverage criterion
  definition, 17
coverage level
  definition, 18
criteria
  ACoC, 160, 163
  ADC, 48, 50, 70, 79, 102
  ADUPC, 48–51, 70, 102, 103
  AUC, 48–51, 70, 79, 102, 271–272
  BCC, 162–163
  CACC, 109–111, 113, 115, 118–119, 126–129, 133, 134, 142, 147–148, 188, 272
  CC, 106, 113, 116–117, 126, 132, 187
  CoC, 107, 113, 117
  CPC, 36, 50
  CRTC, 36, 50
  CUTPNFP, 113, 142–149
  DC, 172
  EC, 34, 50–51, 54, 65–66, 79, 90, 96, 101, 172, 187, 268–271
  ECC, 160–161, 163
  EPC, 35, 50, 79
  GACC, 109, 113, 117–119, 134, 142, 147, 148, 188
  GICC, 112
  IC, 113, 139–141, 149
  MBCC, 162–163
  MC, 175
  MOC, 175
  MPC, 175
  NC, 33–35, 50, 54, 65–67, 78, 87, 90, 96, 101, 172, 186, 268–271
  PC, 106–109, 112, 113, 115, 116, 118–120, 125, 128–129, 132, 134, 140
  PDC, 172
  PPC, 35–36, 39, 50–51
  PWC, 161, 163
  RACC, 110–111, 113, 115, 118–119, 134, 142, 144, 146–148, 188

criteria (*cont.*)
  RICC, 112, 113
  SMC, 178
  SPC, 36, 90, 96
  SRTC, 36, 50
  TSC, 172
  TWC, 161–163
  UTPC, 113, 140–146
  WMC, 179
criteria subsumption, *see* subsumption
criterion
  definition, 17
  explanation, 17–18

debugging
  definition, 13
def, 44–45
  coupling-def, 247
  first-use, 247
  last-def, 72, 247
    definition, 68
    example, 69–70
def-clear, 45, 247
def-pair set, 46
def-path set, 45
deploy, 287
detailed design, 6, 223
determination
  definition, 108
  examples, 108–111, 116–119, 126–127, 132, 134–146
  explanation, 107, 113–115
detour, 39
  definition, 38
  explanation, 38–39
Disjunctive Normal Form (DNF), 138
driver, 218
  definition, 218
du-pair, 44, 57–59, 69–70
  interprocedural, 72–73
du-path, 45–49, 51, 57–59, 69–70
dynamic testing
  definition, 22

economics, 286–287
edge, 27–30
embedded, 262–265, 287–288
emergent, 280–283
error
  definition, 12
  examples, 7–8, 12–13
  explanation, 12
  timeliness, 264
example Java
  cal(), 132, 190
  checkIt(), 130
  countPositive(), 16
  findLast(), 16
  findVal(), 189
  lastZero(), 16
  numZero(), 12
  oddOrPos(), 16
  power(), 190
  Quadratic, 71
  Queue, 85
  Stutter, 79
  takeOut(), 74
  TestPat, 56
  trash(), 74
  TriTyp, 121, 152–163
  twoPred(), 130
  union(), 10
executable test script
  definition, 15
exit commands
  definition, 15
expected results
  definition, 14

failure
  definition, 12
  examples, 7–8, 12–13
  explanation, 12
  timeliness, 264
fault, 142, 288
  definition, 12
  examples, 7–8, 12–13
  explanation, 12
  specific types, 142
  timeliness, 263
first-use, 27, *see* use, first-use

generator, 19, 171
  definition, 18
genetic algorithms, 290
Goldilocks problem, 216
grammar, 170–172
  ground string, 173
  nonterminal, 171
  production, 171
  rule, 171
  start symbol, 171
  terminal, 171
graph, 27–52
  case structure, 54
  double-diamond, 30–91
  for structure, 54
  if structure, 53
  if-else structure, 52
  SESE, 30
  while structure, 53
graph coverage
  definition, 33
Graphical User Interface, 260–262

implementation, 6
implicant, 138
  prime, 140
  redundant, 140
inactive clause coverage (ICC)
  ambiguity, 111–112
  definition, 111
  examples, 112
infeasible, 36, 39, 59, 77, 103, 112–113, 165
  CACC and RACC, 110
  subsumption, 20
  test requirements, 18
infection, 13, 178, 284–285
input
  invalid, 173, 261, 283

usage distribution, 285
valid, 173
input domain, 150, 152
model, 151, 152
Input Domain Model (IDM), 152–158
constraints, 165
instrumentation, 268–272
data flow coverage, 271–272
definition, 268
edge coverage, 270–271
logic coverage, 272
node coverage, 268–270
intermediate design, 222

Java bytecode, 270, 276–277
Java reflection, 270, 276–277
jelly bean, 17–20

Karnaugh maps, 140, 144–146

last-def, *see* def, last-def
literal, 138

maintenance changes, 216
adaptive, 216
corrective, 216
perfective, 216
preventive, 216
major clause, 107–115, 126–127
definition, 107
Mars lander, 8
minor clause, 107–115, 126–127
definition, 107
misuse case, 283
mock
definition, 218
MSG, *see* mutation, schema-based
muJava, 277
mutation
adequacy, 181
bytecode translation, 277
dead, 177, 180
effective operators, 182
equivalent, 177
interpretation, 274
java reflection, 276–277
kill, 175, 177
mutant, 173
operator, 173, 182–185
real-time, 265
schema-based, 274–277
score, 175
selective, 182
separate compilation, 274–275
SMV, 198–201
specification, 198–201
stillborn, 177
strong, 178–180
strongly kill, 178
tool building, 272–277
trivial, 177
weak, 178–180, 186
weakly killing, 179
XML, 203–204

near false point, 142
node, 27–30
final, 27
initial, 27
node coverage
definition, 33, 34

object-oriented
class, 236
data abstraction, 236
data flow testing, 247–256
dynamic binding, 236
inheritance, 236
inheritance fault, 240–247
inter-class testing, 237
inter-method testing, 237
intra-class testing, 237
intra-method testing, 237
override, 238
polymorphism, 236, 238
subclass inheritance, 236
substitution principle, 236
subtype inheritance, 236
testing, 236–256
yo-yo graph, 239–240
observability, 263, 284–286
definition, 14
oracle, 230–231
consistency check, 231–232
data redundancy, 232–233
definition, 230
direct verification, 230
redundant computation, 231

partition, 150, 153
complete, 151–152, 155, 158
disjoint, 150–152, 155, 158
path, 29–32
du-path, *see* du-path
prime, 51
definition, 35
deriving, 39–42
examples, 37
simple, 35–36, 39, 51
test-path, 30
path expressions, 91, 93–96, 99
path product, 91
Pentium, 7
postfix values
definition, 15
predicate, 106
definition, 104
examples, 105
minimal, 140
prefix values
definition, 15
prime path, *see* path, prime
propagation, 13, 178, 284–285
proper subterm, 140

quality assurance, 225–226

reachability, 13, 178, 284–285
real-time, 262–265, 268
environment, 262

real-time (*cont.*)
  reactive, 262
  response time, 263
  tasks, 262
  timeliness, 262, 264
recognizer, 19, 171
  definition, 18
regression test
  inclusive, 217
  modification-revealing, 217
  precise, 217
regular expression, 170–171, 201–202
regular expressions, 91, 94–96
reproducibility, 263
requirements analysis, 4–221
RIP model, 13, 178, 284
round trip, 36

safety, 280–284, 287
safety constraints, 135
security, 204, 205, 235, 257, 280–284, 288
SESE graph, *see* graph, SESE
sidetrip, 39, 103
  definition, 38
  explanation, 38–39
simple path, *see* path, simple
static testing
  definition, 22
stub, 218
  definition, 218
subgraph, 27
subpath, 29
subsumption, 19, 21, 23, 34, 50, 112–113, 142–144, 163
  ACC, IC and UTPC, 141–142
  clause and predicate coverage, 106–107
  definition, 19
  explanation, 19–20, 289
  graph, 142
  infeasible, 20
  mutation, 186–189
  partitioning criteria, 163
  predicate coverage and ACC, 109, 129
  predicate coverage and IC, 140
subsystem design, 6
system and software design, 221–222

term, 138
test action, 220–223
test case
  definition, 15
test case values
  definition, 14
test design, 220–223
test driver, *see* driver
test failure
  definition, 13
test influence, 220–223
test path, *see* path, test-path
test plan, 225–230
  mission, 226
  strategic, 226
  tactical, 226
test process, 219–220
test requirement
  definition, 17
  explanation, 17–18
test set
  definition, 15
test stub, *see* stub
testability, 222, 284–286
testing
  acceptance, 6
  definition, 13
  deployment, 220, 224
  integration, 6, 217–218, 220, 222–224, 288
    definition, 217
  module, 6, 218, 288
  regression, 215–217, 224–225, 231
    definition, 215
  system, 6, 222, 288
  unit, 6, 218, 222–224, 288
top-down testing
  definition, 22
tour, 31, 32, 39, 70
  best effort
    definition, 39
  definition, 38
  explanation, 38–39
Traffic Collision and Avoidance System (TCAS), 232–233
transaction flow graph, 89

unique true point, 140
usability testing, 260
use, 44–45
  coupling, 247
  first-use, 72
    definition, 69
    example, 69–70
use case, 87–90

validation
  definition, 11
verification
  definition, 11
verification values
  definition, 15
version control, 215, 225
visit, 31, 32

web, 287
  client-side, 257–258
  dynamic page, 256
  server-side, 258–259
  site, 256
  static page, 256
  static site, 256
  web application, 256–259, 287
    test case, 257
  web service, 256–260
white-box testing
  definition, 21

FOUNDATIONS OF

SOFTWARE TESTING

2E

ADITYA P. MATHUR

ALWAYS LEARNING

PEARSON

Foundations of Software Testing

SECOND EDITION

Fundamental Alogorithms and Techniques

An Undergraduate and Graduate Text
A Reference for the Practioning Software Engineer

ADITYA P. MATHUR
Purdue University

PEARSON

Delhi • Chennai

Copyright © 2013 Dorling Kindersley (India) Pvt. Ltd.
Licensees of Pearson Education in South Asia

No part of this eBook may be used or reproduced in any manner whatsoever without the publisher's prior written consent.

This eBook may or may not include all assets that were part of the print version. The publisher reserves the right to remove any material in this eBook at any time.

ISBN 9788131794760
eISBN 9789332517653

Head Office: A-8(A), Sector 62, Knowledge Boulevard, 7th Floor, NOIDA 201 309, India
Registered Office: 11 Local Shopping Centre, Panchsheel Park, New Delhi 110 017, India

In Memory of

My Loving Mother and Father

My Friend (Late) Dr. Ranjit Singh Chauhan

This page is intentionally left blank.

Contents

Preface to the Second Edition xiii

Preface to the First Edition xix

Acknowledgements xxvii

PART I: PRELIMINARIES

1. PRELIMINARIES: SOFTWARE TESTING 3

1.1. HUMANS, ERRORS, AND TESTING 4

1.1.1 Errors, faults, and failures 5

1.1.2 Test automation 7

1.1.3 Developer and tester as two roles 7

1.2. SOFTWARE QUALITY 8

1.2.1 Quality attributes 8

1.2.2 Reliability 10

1.3. REQUIREMENTS, BEHAVIOR, AND CORRECTNESS 11

1.3.1 Input domain 12

1.3.2 Specifying program behavior 13

1.3.3 Valid and invalid inputs 17

1.4. CORRECTNESS VERSUS RELIABILITY 18

1.4.1 Correctness 18

1.4.2 Reliability 19

1.4.3 Operational profiles 20

1.5. TESTING AND DEBUGGING 21

1.5.1 Preparing a test plan 21

1.5.2 Constructing test data 21

1.5.3 Executing the program 23

1.5.4 Assessing program correctness 25

1.5.5 Constructing an oracle 26

1.6. TEST METRICS 28

1.6.1 Organizational metrics 28

1.6.2 Project metrics 29

1.6.3 Process metrics 30

1.6.4 Product metrics: generic 30

1.6.5 Product metrics: OO software 32

1.6.6 Progress monitoring and trends 33

1.6.7 Static and dynamic metrics 33

1.6.8 Testability 33

1.7. SOFTWARE AND HARDWARE TESTING 35

1.8. TESTING AND VERIFICATION 37

1.9. DEFECT MANAGEMENT 38

1.10. TEST GENERATION STRATEGIES 39

1.11. STATIC TESTING 42

1.11.1 Walkthroughs 42

1.11.2 Inspections 43

1.11.3 Software complexity and static testing 43

1.12. MODEL-BASED TESTING AND MODEL CHECKING 44

1.13. TYPES OF TESTING 45

1.13.1 Classifier: C1: Source of test generation 47

1.13.2 Classifier: C2: Life cycle phase 48

1.13.3 Classifier: C3: Goal-directed testing 50

1.13.4 Classifier: C4: Artifact under test 53

1.13.5 Classifier: C5: Test process models 54
1.14. THE SATURATION EFFECT 59
1.14.1 Confidence and true reliability 60
1.14.2 Saturation region 60
1.14.3 False sense of confidence 61
1.14.4 Reducing Δ 61
1.14.5 Impact on test process 63
1.15. PRINCIPLES OF TESTING 63
1.16. Tools 66
Summary 67
Exercises 68

2. PRELIMINARIES: MATHEMATICAL 71

2.1. PREDICATES AND BOOLEAN EXPRESSIONS 72
2.2. CONTROL FLOW GRAPH 74
2.2.1 Basic blocks 74
2.2.2 Flow graphs 76
2.2.3 Paths 77
2.2.4 Basis paths 83
2.2.5 Path conditions and domains 85
2.2.6 Domain and computation errors 90
2.2.7 Static code analysis tools and static testing 91
2.3. EXECUTION HISTORY 92
2.4. DOMINATORS AND POST-DOMINATORS 93
2.5. PROGRAM DEPENDENCE GRAPH 95
2.5.1 Data dependence 95
2.5.2 Control dependence 96
2.5.3 Call graph 97
2.6. STRINGS, LANGUAGES, AND REGULAR EXPRESSIONS 98
2.7. TOOLS 99
Summary 101
Exercises 101

PART II: TEST GENERATION

3. DOMAIN PARTITIONING 105

3.1. INTRODUCTION 106
3.2. THE TEST SELECTION PROBLEM 107
3.3. EQUIVALENCE PARTITIONING 109
3.3.1 Faults targeted 110
3.3.2 Relations 111
3.3.3 Equivalence classes for variables 116
3.3.4 Unidimensional partitioning versus multi-dimensional partitioning 121
3.3.5 A systematic procedure 123
3.3.6 Test selection 128
3.3.7 Impact of GUI design 131
3.4. BOUNDARY VALUE ANALYSIS 133
3.5. CATEGORY-PARTITION METHOD 140
Summary 147
Exercises 147

4. PREDICATE ANALYSIS 152

4.1. INTRODUCTION 153
4.2. DOMAIN TESTING 153
4.2.1 Domain errors 153
4.2.2 Border shifts 154
4.2.3 ON-OFF points 155
4.2.4 Undetected errors 157
4.2.5 Coincidental correctness 159
4.2.6 Paths to be tested 159
4.3. CAUSE-EFFECT GRAPHING 160
4.3.1 Notation used in cause-effect graphing 161
4.3.2 Creating cause-effect graphs 163
4.3.3 Decision table from cause-effect graph 167
4.3.4 Heuristics to avoid combinatorial explosion 172

4.3.5 Test generation from a decision table 175

4.4. TESTS USING PREDICATE SYNTAX 176

4.4.1 A fault model 176

4.4.2 Predicate constraints 179

4.4.3 Predicate testing criteria 180

4.4.4 BOR, BRO, and BRE adequate tests 182

4.4.5 BOR constraints for non-singular expressions 191

4.4.6 Cause-effect graphs and predicate testing 197

4.4.7 Fault propagation 198

4.4.8 Predicate testing in practice 200

4.5. TESTS USING BASIS PATHS 204

4.6. SCENARIOS AND TESTS 205

Summary 207

Exercises 207

5. TEST GENERATION FROM FINITE STATE MODELS 213

5.1. SOFTWARE DESIGN AND TESTING 214

5.2. FINITE STATE MACHINES 216

5.2.1 Excitation using an input sequence 220

5.2.2 Tabular representation 221

5.2.3 Properties of FSM 222

5.3. CONFORMANCE TESTING 224

5.3.1 Reset inputs 226

5.3.2 The testing problem 228

5.4. A FAULT MODEL 228

5.4.1 Mutants of FSMs 231

5.4.2 Fault coverage 233

5.5. CHARACTERIZATION SET 234

5.5.1 Construction of the k-equivalence partitions 235

5.5.2 Deriving the characterization set 239

5.5.3 Identification sets 241

5.6. THE W-METHOD 242

5.6.1 Assumptions 242

5.6.2 Maximum number of states 243

5.6.3 Computation of the transition cover set 243

5.6.4 Constructing Z 245

5.6.5 Deriving a test set 245

5.6.6 Testing using the W-method 246

5.6.7 The error detection process 249

5.7. THE PARTIAL W-METHOD 250

5.7.1 Testing using the Wp-method for $m = n$ 252

5.7.2 Testing using the Wp-method for $m > n$ 254

5.8. THE UIO-SEQUENCE METHOD 257

5.8.1 Assumptions 257

5.8.2 UIO sequences 257

5.8.3 Core and non-core behavior 259

5.8.4 Generation of UIO sequences 261

5.8.5 Distinguishing signatures 273

5.8.6 Test generation 275

5.8.7 Test optimization 277

5.8.8 Fault detection 279

5.9. AUTOMATA THEORETIC VERSUS CONTROL-FLOW BASED TECHNIQUES 282

5.9.1 n-switch-cover 285

5.9.2 Comparing automata theoretic methods 286

5.10. TOOLS 287

Summary 288

Exercises 289

6. TEST GENERATION FROM COMBINATORIAL DESIGNS 295

6.1. COMBINATORIAL DESIGNS 296

6.1.1 Test configuration and test set 297

6.1.2 Modeling the input and configuration spaces 297

6.2. A COMBINATORIAL TEST DESIGN PROCESS 303
6.3. FAULT MODEL 305
6.4. LATIN SQUARES 308
6.5. MUTUALLY ORTHOGONAL LATIN SQUARES 310
6.6. PAIRWISE DESIGN: BINARY FACTORS 312
6.7. PAIRWISE DESIGN: MULTI-VALUED FACTORS 318
6.8. ORTHOGONAL ARRAYS 326
6.9. COVERING AND MIXED-LEVEL COVERING ARRAYS 330
6.10. ARRAYS OF STRENGTH > 2 332
6.11. GENERATING COVERING ARRAYS 333
6.12. TOOLS 342
Summary 342
Exercises 343

PART III: TEST ADEQUACY ASSESSMENT AND ENHANCEMENT

7. TEST ADEQUACY ASSESSMENT USING CONTROL FLOW AND DATA FLOW 349

7.1. TEST ADEQUACY: BASICS 350
7.1.1 What is test adequacy? 350
7.1.2 Measurement of test adequacy 351
7.1.3 Test enhancement using measurements of adequacy 353
7.1.4 Infeasibility and test adequacy 358
7.1.5 Error detection and test enhancement 360
7.1.6 Single and multiple executions 363
7.2. ADEQUACY CRITERIA BASED ON CONTROL FLOW 364
7.2.1 Statement and block coverage 364
7.2.2 Conditions and decisions 367
7.2.3 Decision coverage 369
7.2.4 Condition coverage 371
7.2.5 Condition/decision coverage 373
7.2.6 Multiple condition coverage 375
7.2.7 Linear code sequence and jump (LCSAJ) coverage 378
7.2.8 Modified condition/decision coverage 382
7.2.9 MC/DC adequate tests for compound conditions 383
7.2.10 Definition of MC/DC coverage 387
7.2.11 Minimal MC/DC tests 396
7.2.12 Error detection and MC/DC adequacy 396
7.2.13 Short-circuit evaluation and infeasibility 398
7.2.14 Basis path coverage 400
7.2.15 Tracing test cases to requirements 400
7.3. CONCEPTS FROM DATA FLOW 402
7.3.1 Definitions and uses 404
7.3.2 C-use and p-use 405
7.3.3 Global and local definitions and uses 405
7.3.4 Data flow graph 406
7.3.5 Def-clear paths 408
7.3.6 Def-use pairs 409
7.3.7 Def-use chains 410
7.3.8 A little optimization 411
7.3.9 Data contexts and ordered data contexts 412
7.4. ADEQUACY CRITERIA BASED ON DATA FLOW 415
7.4.1 c-use *coverage* 416
7.4.2 p-use *coverage* 417
7.4.3 All-uses *coverage* 419
7.4.4 k-dr *chain coverage* 419

7.4.5 *Using the* k-dr *chain coverage* 421
7.4.6 *Infeasible c- and p-uses* 422
7.4.7 *Context coverage* 423
7.5. Control flow versus data flow 425
7.6. The "subsumes" relation 427
7.7. Structural and functional testing 429
7.8. Scalability of coverage measurement 431
7.9. Tools 433
Summary 434
Exercises 435

8. Test Adequacy Assessment using Program Mutation 442
8.1. Introduction 443
8.2. Mutation and mutants 444
8.2.1 *First-order and higher order mutants* 445
8.2.2 *Syntax and semantics of mutants* 446
8.2.3 *Strong and weak mutations* 449
8.2.4. *Why mutate?* 450
8.3. Test assessment using mutation 452
8.3.1 *A procedure for test adequacy assessment* 452
8.3.2 *Alternate procedures for test adequacy assessment* 462
8.3.3 *"Distinguished" versus "killed" mutants* 463
8.3.4 *Conditions for distinguishing a mutant* 463
8.4. Mutation operators 465
8.4.1 *Operator types* 467
8.4.2 *Language dependence of mutation operators* 468
8.5. Design of mutation operators 470
8.5.1 *Goodness criteria for mutation operators* 470
8.5.2 *Guidelines* 471
8.6. Founding principles of mutation testing 472
8.6.1 *The competent programmer hypothesis* 472
8.6.2 *The coupling effect* 473
8.7. Equivalent mutants 474
8.8. Fault detection using mutation 474
8.9. Types of mutants 477
8.10. Mutation operators for C 478
8.10.1. *What is not mutated?* 479
8.10.2 *Linearization* 480
8.10.3 *Execution sequence* 481
8.10.4 *Effect of an execution sequence* 484
8.10.5 *Global and local identifier sets* 485
8.10.6 *Global and local reference sets* 486
8.10.7 *Mutating program constants* 489
8.10.8 *Mutating operators* 490
8.10.9 *Mutating statements* 497
8.10.10 *Mutating program variables* 511
8.11. Mutation operators for Java 516
8.11.1 *Traditional mutation operators* 517
8.11.2 *Inheritence* 518
8.11.3 *Polymorphism and dynamic binding* 521
8.11.4 *Method overloading* 523
8.11.5 *Java specific mutation operators* 524
8.12. Comparison of mutation operators 526
8.13. Mutation testing within budget 528

8.13.1 Prioritizing functions to be mutated 529
8.13.2 Selecting a subset of mutation operators 530
8.14. Case and program testing 531
8.15. Tools 533
Summary 535
Exercises 536

PART IV: PHASES OF TESTING

9. Test Selection, Minimization, and Prioritization for Regression Testing 547
9.1. What is regression testing? 548
9.2. Regression test process 549
9.2.1 Revalidation, selection, minimization, and prioritization 550
9.2.2 Test setup 551
9.2.3 Test sequencing 552
9.2.4 Test execution 554
9.2.5 Output comparison 555
9.3. Regression test selection: the problem 555
9.4. Selecting regression tests 557
9.4.1 Test all 557
9.4.2 Random selection 557
9.4.3 Selecting modification traversing tests 557
9.4.4 Test minimization 558
9.4.5 Test prioritization 559
9.5. Test selection using execution trace 560
9.5.1 Obtaining the execution trace 560
9.5.2 Selecting regression tests 562
9.5.3 Handling function calls 567
9.5.4 Handling changes in declarations 567
9.6. Test selection using dynamic slicing 570
9.6.1 Dynamic slicing 572
9.6.2 Computation of dynamic slices 572
9.6.3 Selecting tests 574
9.6.4 Potential dependence 575
9.6.5 Computing the relevant slice 579
9.6.6 Addition and deletion of statements 580
9.6.7 Identifying variables for slicing 581
9.6.8 Reduced dynamic dependence graph 582
9.7. Scalability of test selection algorithms 583
9.8. Test minimization 586
9.8.1 The set cover problem 587
9.8.2 A procedure for test minimization 587
9.9. Test prioritization 590
9.10. Tools 594
Summary 595
Exercises 596

10. Unit Testing 601
10.1. Introduction 602
10.2. Context 602
10.3. Test design 603
10.4. Using junit 605
10.5. Stubs and mocks 609
10.5.1 Using mock objects 609
10.6. Tools 616
Summary 617
Exercises 617

11. Integration Testing 619

11.1. Introduction 620

11.2. Integration errors 621

11.3. Dependence 622

11.3.1 Class relationships: static 625

11.3.2 Class relationships: dynamic 626

11.3.3 Class firewalls 627

11.3.4 Precise and imprecise relationships 628

11.4. OO Versus non-OO programs 630

11.5. Integration hierarchy 631

11.5.1 Choosing an integration strategy 633

11.5.2 Comparing integration strategies 634

11.5.3 Specific stubs and retesting 635

11.6. Finding a near-optimal test order 637

11.6.1 The TD method 637

11.6.2 The TJJM method 649

11.6.3 The BLW method 654

11.6.4 Comparison of TD, TJJM, and the BLW methods 658

11.6.5 Which test order algorithm to select? 661

11.7. Test generation 662

11.7.1 Data variety 662

11.7.2 Data constraints 664

11.8. Test assessment 665

11.9. Tools 667

Summary 668

Exercises 669

Index 673

This page is intentionally left blank.

Preface to the Second Edition

Welcome to the Second Edition of *Foundations of Software Testing*. Thanks to the many instructors, students, and professionals, the wide adoption of the first edition, and comments and suggestions from the readers, have led to this thoroughly revised version of the first edition. Changes and additions to the text are summarized next.

Reorganization

The second edition is divided into four parts. Part I, titled "Preliminaries," contains two chapters. Chapter 1 introduces the basics of software testing and Chapter 2 some mathematical preliminaries. Part II, titled "Test Generation," contains four chapters. Chapter 3 introduces test generation methods based on partitioning the input and output domains of the program under test. Chapter 4 contains test generation methods based on an analysis of the predicates extracted from the requirements or found in the code. Chapter 5 introduces techniques for generating tests from finite state models of a program. Such models are usually derived from the requirements and constitute an element of the overall program design. Chapter 6 introduces techniques for generating tests using various combinatorial designs. Part III, titled "Test Adequacy and Enhancement," contains two chapters. Chapter 7 introduces control and data flow-based criteria for the measurement of test adequacy. Chapter 8 is a comprehensive introduction to program mutation also as a method for assessing test adequacy. Finally, Part IV, titled "Phases of Testing," contains three chapters. Chapter 9 introduces fundamentals of regression testing. Chapter 10 introduces techniques and tools for unit testing. And lastly, Chapter 11 introduces various techniques useful during integration testing.

New Material

Readers used to the first edition will find a significant amount of new material in the second edition. Selection of new topics was based on requests from the readers. Unfortunately not all requests for additional material could be entertained in order to keep the size and cost of the book from exploding to an unacceptable level.

Tools: All chapters starting at Chapter 5 and onwards include a section titled "Tools." This section lists tools available that employ some of the techniques discussed in the corresponding chapter. In Chapter 10, a few tools are also described in some detail. Indeed, this is not a book on tools and hence these sections are kept relatively small. The purpose of these sections is to list certain tools to the readers from which one or more could be selected. Of course, software testing tools undergo change and some become obsolete. An attempt has been made to include only those tools that were available either freely or commercially at the time of writing this edition.

Part I: A new subsection, titled "Principles of Testing," has been added to Chapter 1. While the literature in software testing contains a well-known and well-written paper by Bertrand Meyers, the material in this section is solely that of the author. The principles enumerated here are based on the author's experience with, and his knowledge of, software testing. Also, there is new material on basis paths in Chapter 2.

Part II: Chapter 2 from the first edition has now been split into two shorter chapters. Domain testing using the notion of ON-OFF points is added to Chapter 4. Test generation using basis paths introduced in Chapter 2 has also been included in Chapter 4.

Part III: Removal of errors is the only change in this part of the book.

Part IV: This part contains three chapters corresponding one each to the three phases of the software development cycle, namely, regression testing, unit testing, and integration testing. Chapter 9 remains unchanged from the first edition. Chapter 10 introduces unit testing and the commonly used tools such as JUnit. The topic of stubs and mocks is covered here. Chapter 11 introduces some problems that arise in integration testing and methods to solve them. Finding a near-optimal test order is a well-known problem in integration testing. Three algorithms to solve this problem are described in detail. These are compared using randomly generated Object Relation Diagrams (ORD). Each algorithm was coded in Java and the code is freely available at the Web site maintained by the author—http://www.cs.purdue.edu/homes/apm/foundationsBook/.

Solutions to Exercises

Professor Vahid Garousi at the University of Calgary maintains a list of solutions to selected exercises found at the end of the chapters in the first edition. It is my hope that using "reader sourcing," we will be able to obtain solutions to all exercises in the second edition.

Errors

Several readers pointed to errors in the first edition. Some of these are technical and some editorial. All errors reported have been corrected in the second edition. A complete list of errors removed is available at http://www.cs.purdue.edu/homes/apm/foundationsBook/errata.html. Errors reported in the second edition will also be available at this site.

Suggestions to instructors

Instructors who are satisfied with the first edition will find that the material they used is available in the second edition. Hence, they need not make any changes in their curriculum. However, some instructors might wish to add the new material to their classes. Given the importance of unit testing, it might be best to introduce this subject towards the beginning of a course based on this book. Similarly, the material on domain testing can be introduced when the basic techniques for test generation are covered. The material on integration testing might be better suited for a graduate course in software testing.

Table 1 lists a sample weekly program for an undergraduate course in software testing. This sample is different from that given in Table 1 in the first edition of this book. However, it is important that students be exposed to a variety of testing tools. Experience indicates that most students learn best when working in small groups. Hence, it is perhaps best to create several small teams of students and assign them the task of learning to use

Table 1 Sample weekly program for a 17-week undergraduate course in software testing.

Week	Chapter	Comments
1	1: Preliminaries	Form groups of 3 to 5 and assign each group a set of about 5 software testing tools to study on their own. Each group should be asked to make a presentation of one tool starting week 3. The actual presentation schedule will depend on the number of student groups.
2	2: Mathematical preliminaries	Give one short quiz during each week starting in week 2.

Table 1 (Continued)

Week	Chapter	Comments
3	10: Unit testing	Assign exercises using JUnit or a similar unit testing tool.
4	3: Domain partitioning	
5	3: Domain partitioning	
6	4: Predicate analysis	
7	4: Predicate analysis	
8	5: Test generation: FSM models; review	 Midterm exam
9	5: Test generation: FSM models	
10	5: Test generation: FSM models	
11	6: Combinatorial designs	
12	6: Combinatorial designs	Introduce at least one tool for test generation using combinatorial designs.
13	7: Control and data flow	
14	7: Control and data flow	
15	7: Control and data flow	
16	9: Regression testing	
17	Review	Final exam

a small set of, say, five freely available testing tools. Each team can demonstrate to the entire class the tools they have been assigned. This enables the entire class to be exposed to a variety of testing tools.

In addition to the testing tools, it is also important that teams of students engage in term projects. Such projects could be sponsored by commercial firms or by the instructor. Experience indicates that students enjoy working on company-sponsored projects and learn a lot about the challenges of software testing.

Cash awards have been given in the past to those who found and reported errors. This offer will continue for the second edition. Please visit the book's Web site for details.

Aditya P. Mathur

This page is intentionally left blank.

Preface to the First Edition

Welcome to *Foundations of Software Testing*! This book is intended to offer exactly what its title implies. It is important that students planning a career in Information Technology take a course in software testing. It is also important that such a course offer students an opportunity to acquire material that will remain useful throughout their careers in a variety of software applications and changing environments. This book is intended to be an introduction to exactly such material and hence ideal as text for a course in software testing. It distills knowledge developed by hundreds of testing researchers and practitioners from all over the world and brings it to its readers in easy to understand form.

Test generation, selection, prioritization, and assessment lie at the foundation of all technical activities involved in software testing. Appropriate deployment of the elements of this strong foundation enables the testing of different types of software applications as well as testing for various properties. Applications include OO systems, web services, Graphical User Interfaces, embedded systems, and others. Properties relate to security, timing, performance, reliability, and others.

The importance of software testing increases as software pervades more and more of our daily lives. Unfortunately, few universities offer full-fledged courses in software testing. Those that do often struggle to identify a suitable text. My hope is that this book will allow academic institutions to create courses in software testing, and those that already offer such courses will not need to hunt for a textbook or rely solely on research publications.

Conversations with testers and managers in commercial software development environments have led me to believe that though software testing is considered an important activity, software testers often complain of not receiving treatment at par with system designers and developers. I believe that raising the level of sophistication in the material covered in courses in software testing will lead to superior testing practices, high quality software, and thus translate into positive impact on the career of software testers. I hope that exposure to even one-half of the material in Volume 1 will establish a student's respect for software testing as a discipline in its

own right and at the same level of maturity as subjects such as Compilers, Databases, Algorithms, and Networks.

Target Audience

It is natural to ask: What is the target level of this book? My experience, and that of some instructors who have used earlier drafts, indicates that Volume 1 is best suited for use at senior undergraduate and early graduate levels. Chapters in Volume 2 are appropriate for a course at the graduate level and make it even suitable for a third advanced course in software testing. Some instructors might want to include material from Volume 2, such as security testing, in an undergraduate course. Certainly, no single one-semester course can hope to cover all material in either of the two volumes without drowning the students! However, a carefully chosen subset of chapters can offer an intellectually rich and useful body of material and prepare the students for a rewarding career in software development and testing.

While the presentation in this book is aimed at a student in a college or university classroom, I believe that both practitioners and researches will find it useful. Practitioners, with patience, may find this book a rich source of techniques they could learn and adapt in their development and test environment. Researchers will likely find this book as a rich reference.

Nature of Material Covered

Software testing covers a wide spectrum of activities. At a higher level, such activities appear to be similar whereas at a lower level of detail they might differ significantly. For example, most software development environments engage in test execution. However, test execution for an operating system is carried out quite differently than that for a pacemaker; while one is an open-system, the other is embedded and hence the need for different ways to execute tests.

The simultaneous existence of similarities and differences in each software testing activity leads to a dilemma for an author as well as an instructor. Should a book, and a course, focus on specific software development environments and how they carry out various testing activities? Or, should they focus on specific testing activities without any detailed recourse to specific environments? Either strategy is subject to criticism and leaves the student in a vacuum regarding the applications of testing activities or about their formal foundations.

I have resolved this dilemma through careful selection and organization of the material presented. Parts I, II, and III of the book focus primarily on the foundations of various testing activities while Part IV focuses on the applications of the foundational material. For example, techniques for generating tests from models of expected program behavior are covered in Part II while the application of these techniques to testing object-oriented programs and for security properties are covered in Part IV. As an exception, Part I does illustrate through examples the differences in software test process as applied in various software development organizations.

Organization

The book is organized into four parts, each distributed over two volumes. Part I covers terminology and preliminary concepts related to software testing. These are divided into three chapters one of which, Chapter 1, appears in Volume 1 while the remaining two on errors and test process will be included in Volume 2. Chapter 1 introduces a variety of terms and basic concepts that pervade the field of software testing. Some of the early adopters of this book use Chapter 1 for introductory material covered during the first two or three weeks of an undergraduate course.

Part II covers various test generation techniques. Chapter 2 introduces the most fundamental of all test generation techniques widely applicable in almost any software application one can imagine. These include equivalence partitioning, boundary value analysis, cause-effect graphing, and predicate testing. Chapter 3 introduces powerful and fundamental techniques for automatically generating tests from finite state models. Three techniques have been selected for presentation in this chapter: W, Wp, and UIO methods. Finite state models are used in a variety of applications such as in OO testing, security testing, and GUI testing. Statecharts offer a more powerful modeling formalism than finite state machines. Test generation from statechart models will be covered in Volume 2. Modeling the expected timing behavior of a realtime application using timed automata and generating tests from the model will be covered in Volume 2. Generation of combinatorial designs and tests is the topic of Chapter 4. Generation of tests from formal specifications will be covered in Volume 2. Random, stress, load, and performance testing testing will be covered in Volume 2.

Regression testing forms an integral part of all software development environments where software evolves into newer versions and thus undergoes extensive maintenance. Chapter 5 introduces some fundamental techniques for test selection, prioritization, and minimization of use during regression testing.

Part III is an extensive coverage of an important and widely applicable topic in software testing: test enhancement through measurement of test adequacy. Chapter 6 introduces a variety of control-flow and data-flow based code coverage criteria and explains how these could be used in practice. The most powerful of test adequacy criteria based on program mutation are introduced in Chapter 7. While some form of test adequacy assessment is used in almost every software development organization, material covered in these chapters promises to take adequacy assessment and test enhancement to a new level thereby making a significant positive impact on software reliability.

Practitioners often complain, and are mostly right, that many white-box adequacy criteria are impractical to use during integration and system testing. I have included a discussion on how some of the most powerful adequacy assessment criteria can be, and should be, used even beyond unit testing. Certainly, my suggestions to do so assume the availability of commercial-strength tools for adequacy assessment.

Each chapter ends with a detailed bibliography. I have tried to be as comprehensive as possible in citing works related to the contents of each chapter. I hope that instructors and students will find the bibliography sections rich and helpful in enhancing their knowledge beyond this book. Citations are also a testimony to the rich literature in the field of software testing.

What this Book does not Cover

Software testing consists of a large number of related and intertwined activities. Some of these are technical, some administrative, and some merely routine. Technical activities include test case and oracle design at the unit, subsystem, integration, system, and regression levels. Administrative activities include manpower planning, budgeting, and reporting. Planning activities include test planning, quality assessment and control, and manpower allocation. While some planning activities are best classified as administrative, e.g. manpower allocation, others such as test planning are intertwined with technical activities like test case design.

If you are interested in learning about details of administrative tasks then this is not the right book. While a chapter on test processes in Volume 2 will offer insights into administrative and planning tasks such as quality control, details are best covered in many other excellent books cited in sections titled Bibliographic Notes at the end of each chapter.

Several test related activities are product specific. For example, testing of a device driver often includes tasks such as writing a device simulator. Simulators include heart simulator in testing cardiac pacemakers, a USB

port simulator useful in testing I/O drivers, and an airborne drone simulator used in testing control software for airborne drones. While such activities are extremely important for effective testing and test automation, they often require a significant development effort. For example, writing a device simulator and testing it is both a development and testing activity. Test generation and assessment techniques described in this book are applicable to each of the product specific test activity. However, product specific test activities are illustrated in this book only through examples and not described in any detail. My experience has been that it is best for students to learn about such activities through industry sponsored term projects.

Suggestions to Instructors

There is wide variation in the coverage of topics in courses in software testing. This is one of the reasons why this book is divided into two volumes so as to keep the size and weight of each volume at a comfortable level for the reader while providing a comprehensive treatment of the subject matter. I have tried to cover most, if not all, of the important topics in this area. Tables 2 and 3 provide suggested outline of undergraduate and graduate courses, respectively, that could be based entirely on this book.

Sample Undergraduate Course in Software Testing

We assume a semester long undergraduate course worth 3-credits, that meets twice a week, each meeting lasts 50 minutes, and devotes a total of 17 weeks to lectures, examinations, and project presentations. The course has a 2-hours per week informal laboratory, and requires students to work in small teams of 3 or 4 to complete a term-project. The term project results in a final report and possibly a prototype testing tool. Once every two weeks students are given one laboratory exercise that takes about 4-6 hours to complete.

Table 4 contains a suggested evaluation plan. Carefully designed laboratory exercises form an essential component of this course. Each exercise offers the student an opportunity to use a testing tool to accomplish a task. For example, the objective of a laboratory exercise could be to familiarize the student with JUnit as test runner or JMeter as a tool for the performance measurement of web services. Instructors should be able to design laboratory exercises based on topics covered during the previous weeks. A large number of commercial and open-source testing tools are available for use in a software testing laboratory.

Table 2 A sample undergraduate course in software testing.

Week	Topic	Chapter
1	Course objectives and goals, project assignment, testing terminology and concepts	1
2	Test process and management	1
3	Errors, faults, and failures	1
4	Boundary value analysis, equivalence partitioning, decision tables	2
5, 6	Test generation from predicates	2
7	*Interim project presentations*	
	Review, Midterm examination	
8	Test adequacy: control flow	6
9	Test adequacy: data flow	6
10, 11	Test adequacy: program mutation	7
12, 13, 14	Special topics, e.g. OO testing, security testing	From the lierature
15, 16	Review, *Final project presentations*	
17	*Final examination*	

Sample Graduate Course in Software Testing

We assume a semester long course worth 3-credits. The students entering this course have not had any prior course in software testing, such as the undergraduate course described above. In addition to the examinations, students will be required to read and present recent research material. Students are exposed to testing tools via unscheduled laboratory exercises.

Testing Tools

There is a large set of testing tools available in the commercial, freeware, and open source domains. A small sample of such tools is listed in Table 3.

Evolutionary Book

I expect this book to evolve over time. Advances in topics covered in this book, and any new topics that develop, will be included in subsequent

Table 3 A sample graduate course in software testing.

Week	Topic	Chapter
1	Course objectives and goals, testing terminology and concepts	1
2	Test process and management	To appear in Volume 2
	Errors, faults, and failures	Volume 2
3	Boundary value analysis, equivalence partitioning, decision tables	2
4	Test generation from predicates	2
5, 6	Test generation from finite state models	3
7, 8	Combinatorial designs	4
	Review, Midterm examination	
9	Test adequacy: control flow	6
10	Test adequacy: data flow	6
11, 12	Test adequacy: program mutation	7
13, 14	Special topics, e.g. real-time testing, security testing	From the literature
15, 16	Review, *Research presentations*	
17	*Final examination*	

Table 4 Suggested evaluation components of the undergraduate and graduate courses in software testing.

Level	Component	Weight	Duration
Undergraduate	Midterm examination	15 points	90 minutes
	Final examination	25 points	120 minutes
	Quizzes	10 points	Short duration
	Laboratory assignments	10 points	10 assignments
	Term project	40 points	Semester
Graduate	Midterm examination	20 points	90 minutes
	Final examination	30 points	120 minutes
	Laboratory assignments	10 points	5 assignments
	Research/Term project	30 points	Semester

Table 5 A sample set of tools to select from for use in undergraduate and graduate courses in software testing.

Purpose	Tool	Source
Combinatorial designs	AETG	Telcordia Technologies
Code coverage measurement	TestManager™	IBM Rational
	JUnit	Freeware
	CodeTest	Freescale Semiconductor
	χSuds	Telcordia Technologies
Defect tracking	Bugzilla	Freeware
	FogBugz	Fog Creek Software
GUI testing	WebCorder	Crimson Solutions
	jfcUnit	Freeware
Mutation testing	muJava	Professor Jeff Offut offutt@ise.gmu.edu
	Proteum	Professor Jose Maldonado jcmaldon@icmc.usp.br
Performance testing	Performance Tester	IBM Rational™
	JMeter	Apache, for Java
Regression testing	Eggplant	Redstone Software
	χSuds	Telcordia Technologies
Test management	ClearQuest™	IBM Rational™
	TestManager	IBM Rational™

editions. Any errors found by me and/or reported by the readers will be corrected. The book's web site listed below contains a link to simplify reporting of any errors found. Readers are encouraged to visit the website for latest information about the book.

http://www.cs.purdue.edu/homes/apm/foundationsBook/

In the past I have given cash rewards to students who carefully read the material and reported any kind of error. I plan to retain the cash reward approach as a means for continuous quality improvement.

Aditya P. Mathur

Acknowledgements

A number of people have been of significant assistance in the writing of the first and the second editions. I begin by apologizing to those whose names do not appear here but should have; the omission is purely accidental. My sincere thanks to the many mentioned below.

Rich DeMillo for introducing me to the subject of software testing and supporting my early research. Rich pointed me to the literature that helped with the acquisition of knowledge of and an appreciation of software testing. Bob Horgan who influenced and supported my thinking of the relationship between testing and reliability and the importance of code coverage. Bob freely shared the early and later versions of the χSuds toolset. I continue to believe that χSuds is so far the best toolset available for test adequacy assessment and enhancement. Ronnie Martin for spending endless hours meticulously reviewing many of my technical reports.

Donald Knuth for sharing details of the errors of TEX and to his staff for sharing with me the earlier versions of TEX. Hiralal Agrawal for patiently answering questions related to dynamic slicing. Farokh Bastani, Fevzi Belli, Jim Berger, Kai-Yuan Cai, Ram Chillarege, Sid Dalal, Raymond DeCarlo, Marcio Delamaro, Phyllis Frankl, Arif Ghafoor, Amrit Goel, Dick Hamlet, Mats Heimdahl, Michael A. Hennell, Bill Howden, Ashish Jain, Pankaj Jalote, Rick Karcick, Bogdan Korel, Richard Lipton, Yashwant Malaiya, Josè Maldonado, Simanta Mitra, John Musa, Jeff Offutt, Tom Ostrand, Amit Paradkar, Alberto Pasquini, Ray Paul, C. V. Ramamoorthy, Vernon Rego, Nozer Singpurwalla, Mary-Lou Soffa, Rajesh Subramanian, Kishor Trivedi, Jefferey Voas, Mladen Vouk, Elaine Weyuker, Lee White, and Martin Woodward for discussions and offering constructive criticisms that led to the thrashing of many of my incorrect, often silly, notions about software testing and reliability.

Jim Mapel, Marc Loos, and several other engineers at Boston Scientific Inc. (formerly Guidant Corporation) for introducing me to the sophisticated test processes for ensuring highly reliable cardiac devices. Klaus Diaconu, Mario Garzia, Abdelsalam Heddaya, Jawad Khaki, Nar Ganapathy, Adam Shapiro, Peter Shier, Robin Smith, Amitabh Srivastava, and the many software engineers at Microsoft in the Windows Reliability and Device Driver teams for helping me understand the complex and

sophisticated test processes and tools used at Microsoft to assure the delivery of a highly reliable operating system to millions of people around the globe.

Vahid Garousi has been kind enough to maintain a list of universities that have adopted the first edition of this book as a text. Vahid also maintains a list of solutions to some of the exercises given at the end of each chapter. Please contact Vahid if you are an instructor and wish to obtain a copy of the solutions manual.

Anonymous reviewers for reviewing and providing useful comments. Muhammad Naeem Ayyaz, Abdeslam En-Nouaary, Joao Cangussu, and Eric Wong for editing earlier drafts of this book, and some of the associated presentations, in undergraduate and graduate courses. Feedback from instructors and students in their classes proved invaluable.

David Boardman, Joaõ Cangussu, Mei-Hwa Chen, Byoungju Choi, Praerit Garg, Sudipto Ghosh, Neelam Gupta, Vivek Khandelwal, Edward Krauser, Saileshwar Krishnamurthy, Tsanchi Li, Pietro Michielan, Scott Miller, Manuela Schiona, Baskar Sridharan, Eric Weigman, Eric Wong, and Brandon Wuest for patiently working with me in software testing research.

The following were kind enough to report errors in the very early drafts and the first edition of the book: Emine Gokce Aydal, Fabian Alenius, Christine Ayers, Jordan Fleming, Trupti Gandhi, Vahid Garousi, Lyle Goldman, Nwokedi Idika, K. Jayaram, Yuanlu Jiang, Ashish Kundu, Yu Lei, Yu Leng, Jung-Chi Lin, Shuo Lu, Jim Mapel, Ammar Masood, Kevin McCarthy, Roman Joel Pacheco, Tu Peng, Van Phan, James Roberts, Chetak Sirsat, Kevin Smith, Travis Steel, Marco AurŐlio Graciotto Silva, Praveen Ranjan Srivastava, Natalia Silvis, Bryan Tomayko, Rakesh Veeramacheneni, Yunlin Xu, Hiroshi Yamauchi, Il-Chul Yoon, Eric Wong, Brandon Wuest, Roc Zhang, and Xiangyu Zhang. A very special thank you to Feng Wang who read the first edition cover to cover and reported a number of editorial and technical errors.

Professor T. S. K. V. Iyer for constantly asking me whether or not the book is complete and thus being a major driving force behind its completion. Raymond Miller, Nancy Griffith, Bill Griffith, and Pranas Zunde for their years of support and the welcome I received when I arrived in a new country. John Rice and Elias Houstis for their support in terms of facilities and equipment essential for experimentation and for being wonderful colleagues. Susanne Hambrusch and Ahmed Sameh for supporting me in the development of Software Engineering and Software Testing courses at Purdue. Members of the facilities staff of Computer Science department for assistance with setting up laboratory equipment and software used by students in graduate and undergraduate course offerings at Purdue

University. Patricia Minniear for working hard to make timely copies of drafts of this book for student use.

Professors S. Venkateswaran, L. K. Maheshwari, faculty and staff of the Computer Science Department at BITS, Pilanii, for offering me a friendly work environment during my sabbatical year. My dear friend Mohan Lal and his family for their support over the many years, and especially during my visit to Pilani where I wrote some portions of the first edition. To all the employees of the BITS Guest House (VFAST) whose generosity and love must have had some positive effect on the quality of this book.

Hanna Lena Kovenock who spent countless hours iterating over the cover cartoon depicting the "chair development" team in the animal kingdom. The cartoon did appear on the cover of the draft of the book prior to its publication. Hanna is a great artist and I was fortunate to get her assistance.

My friends Ranjit Gulrajani and Pundi Narasimhan and their families for their emotional support over many years. My wonderful children Gitanjali and Ravishankar for asking "When will the book be in print?" My most beautiful collies Raja and Shaan who now rest permanently, from whom I took away precious play time. My late mother, my father, and mother-in-law Amma for their constant love and support. And last but not the least, my wife Jyoti Iyer Mathur for her constant love and support.

This page is intentionally left blank.

Part I
Preliminaries

Software testing deals with a variety of concepts, some mathematical and others not so mathematical. This first part brings together a set of basic concepts and terminology that pervades software testing. Chapter 1 defines and explains basic terms and concepts that a tester ought to be familiar with. A variety of terms used in the industry, such as test case, test plans, test cycle, and several others, are covered in this chapter. A section on metrics provides an overview of the test metrics used for test process monitoring. Chapter 2 covers a wide range of basic concepts useful in understanding various test generation and test assessment techniques. These include flow graphs, paths, dominators, program dependence, strings, and regular expressions.

This page is intentionally left blank.

1

Preliminaries: Software Testing

CONTENTS

1.1 Humans, errors, and testing
1.2 Software quality
1.3 Requirements, behavior, and correctness
1.4 Correctness versus reliability
1.5 Testing and debugging
1.6 Test metrics
1.7 Software and hardware testing
1.8 Testing and verification
1.9 Defect management
1.10 Test generation strategies
1.11 Static testing
1.12 Model-based testing and model checking
1.13 Types of testing
1.14 The saturation effect
1.15 Principles of testing
1.16 Tools

The purpose of this introductory chapter is to familiarize the reader with the basic concepts related to software testing. Here a framework is set up for the remainder of this book. Specific questions, answered in substantial detail in subsequent chapters, are raised here. After reading this chapter, you are likely to be able to ask meaningful questions related to software testing and reliability.

1.1 Humans, Errors, and Testing

Errors are a part of our daily life.

Errors are a part of our daily life. Humans make errors in their thoughts, in their actions, and in the products that might result from their actions. Errors occur almost everywhere. For example, humans make errors in speech, in medical prescription, in surgery, in driving, in observation, in sports, and certainly in love and software development. Table 1.1 provides examples of human errors. The consequences of human errors vary significantly. An error might be insignificant in that it leads to a gentle friendly smile, such as when a slip of the tongue occurs. Or, an error may lead to a catastrophe, such as when an operator fails to recognize that a relief valve on the pressurizer was stuck open and this resulted in a disastrous radiation leak.

Table 1.1 Examples of errors in various fields of human endeavor.

Area	Error
Hearing	Spoken: He has a garage for repairing *foreign* cars. Heard: He has a garage for repairing *falling* cars.
Medicine	Incorrect antibiotic prescribed.
Music performance	Incorrect note played.
Numerical analysis	Incorrect algorithm for matrix inversion.
Observation	Operator fails to recognize that a relief valve is stuck open.
Software	Operator used: $\neq$, correct operator: $>$. Identifier used: new_line, correct identifier: next_line. Expression used: $a \wedge (b \vee c)$, correct expression: $(a \wedge b) \vee c$. Data conversion from 64-bit floating point to 16-bit integer not protected (resulting in a software exception).
Speech	Spoken: *waple malnut*, intent: *maple walnut*. Spoken: *We need a new refrigerator*, intent: *We need a new washing machine*.
Sports	Incorrect call by the referee in a tennis match.
Writing	Written: What kind of *pans* did you use? Intent: What kind of *pants* did you use?

To determine whether there are any errors in our thought, actions, and the products generated, we resort to the process of *testing*. The primary goal of testing is to determine if the thoughts, actions, and products are as desired, that is, they conform to the requirements. Testing of thoughts is usually designed to determine if a concept or method has been understood satisfactorily. Testing of actions is designed to check if a skill that results in the actions has been acquired satisfactorily. Testing of a product is designed to check if the product behaves as desired. Note that both syntax and semantic errors arise during programming. Given that most modern compilers are able to detect syntactic errors, testing focuses on semantic errors, also known as faults, that cause the program under test to behave incorrectly.

The process of testing offers an opportunity to discover any errors in the product under test.

Example 1.1 An instructor administers a test to determine how well the students have understood what the instructor wanted to convey. A tennis coach administers a test to determine how well the understudy makes a serve. A software developer tests the program developed to determine if it behaves as desired. In each of these three cases there is an attempt by a tester to determine if the human thoughts, actions, and products behave as desired. Behavior that deviates from the desirable is possibly due to an error.

Example 1.2 "Deviation from the expected" may *not* be due to an error for one or more reasons. Suppose that a tester wants to test a program to sort a sequence of integers. The program can sort an input sequence in both descending or ascending orders depending on the request made. Now suppose that the tester wants to check if the program sorts an input sequence in *ascending* order. To do so, she types in an input sequence and a request to sort the sequence in *descending* order. Suppose that the program is correct and produces an output that is the input sequence in descending order.

Upon examination of the output from the program, the tester hypothesizes that the sorting program is incorrect. This is a situation where the tester made a mistake (an error) that led to her incorrect interpretation (perception) of the behavior of the program (the product).

1.1.1 Errors, faults, and failures

There is no widely accepted and precise definition of the term "error." Figure 1.1 illustrates one class of meanings for the terms error, fault, and failure. A programmer writes a program. An *error* occurs in the process

Figure 1.1 Errors, faults, and failures in the process of programming and testing.

of writing a program. A *fault* is the manifestation of one or more errors. A failure occurs when a faulty piece of code is executed leading to an incorrect state that propagates to the program's output. The programmer might misinterpret the requirements and consequently write incorrect code. Upon execution, the program might display behavior that does not match with the expected behavior implying thereby that a *failure* has ocurred. A fault in the program is also commonly referred to as a *bug* or a *defect*. The terms error and bug are by far the most common ways of referring to something "wrong" in the program text that might lead to a failure. In this text, we often use the terms "error" and "fault" as synonyms. Faults are sometimes referred to as *defects*.

In Figure 1.1, notice the separation of "observable" from "observed" behavior. This separation is important because it is the observed behavior

that might lead one to conclude that a program has failed. Certainly, as explained earlier, this conclusion might be incorrect due to one or more reasons.

1.1.2 Test automation

Testing of complex systems, embedded and otherwise, can be a human intensive task. Often one needs to execute thousands of tests to ensure that, for example, a change made to a component of an application does not cause a previously correct code to malfunction. Execution of many tests can be tiring as well error prone. Hence, there is a tremendous need for automating testing tasks.

Test automation aids in reliable and faster completion of routine tasks. However, not all tasks involved in testing are prone to automation.

Most software development organizations automate test related tasks such as regression testing, GUI testing, and I/O device driver testing. Unfortunately the process of test automation cannot be generalized. For example, automating regression tests for an embedded device such as a pacemaker is quite different from that for an I/O device driver that connects to the USB port of a PC. Such lack of generalization often leads to specialized test automation tools developed in-house.

Nevertheless, there do exist general purpose tools for test automation. While such tools might not be applicable in all test environments, they are useful in many. Examples of such tools include Eggplant, Marathon, and Pounder for GUI testing; eLoadExpert, DBMonster, JMeter, Dieseltest, WAPT, LoadRunner, and Grinder for performance or load testing; and Echelon, TestTube, WinRunner, and XTest for regression testing. Despite the existence of a large number and a variety of test automation tools, large software-development organizations develop their own test automation tools primarily due to the unique nature of their test requirements.

AETG is an automated test generator that can be used in a variety of applications. It uses combinatorial design techniques, which we will discuss in Chapter 6. Random testing is often used for the estimation of reliability of products with respect to specific events. For example, one might test an application using randomly generated tests to determine how frequently does it crash or hang. DART is a tool for automatically extracting an interface of a program and generating random tests. While such tools are useful in some environments, they are dependent on the programming language used and the nature of the application interface. Therefore, many organizations develop their own tools for random testing.

1.1.3 Developer and tester as two roles

In the context of software engineering, a developer is one who writes code and a tester is one who tests code. We prefer to treat developer and tester as two distinct but complementary roles. Thus, the same individual could be

A developer is also a tester and vice-versa.

a developer and a tester. It is hard to imagine an individual who assumes the role of a developer but never that of a tester, and vice versa. In fact, it is safe to assume that a developer assumes two roles, that of a "developer" and of a "tester," though at different times. Similarly, a tester also assumes the same two roles but at different times.

Certainly, within a software development organization, the primary role of an individual might be to test and hence this individual assumes the role of a "tester." Similarly, the primarily role of an individual who designs applications and writes code is that of a "developer."

A reference to "tester" in this book refers to the role someone assumes when testing a program. Such an individual could be a developer testing a class she has coded, or a tester who is testing a fully-integrated set of components. A "programmer" in this book refers to an individual who engages in software development and often assumes the role of a tester, at least temporarily. This also implies that the contents of this book are valuable not only to those whose primary role is that of a tester, but also to those who work as developers.

1.2 Software Quality

We all want high-quality software. There exist several definitions of software quality. Also, one quality attribute might be more important to a user than another. In any case, software quality is a multidimensional quantity and is measurable. So, let us look at what defines the quality of a software.

1.2.1 Quality attributes

There exist several measures of software quality. These can be divided into static and dynamic quality attributes. Static quality attributes refer to the actual code and related documentation. Dynamic quality attributes relate to the behavior of the application while in use.

Static quality attributes include structured, maintainable, testable code as well as the availability of correct and complete documentation. You might have come across complaints such as "Product X is excellent, I like the features it offers, but its user manual stinks!" In this case, the user manual brings down the overall product quality. If you are a maintenance engineer and have been assigned the task of doing corrective maintenance on an application code, you will most likely need to understand portions of the code before you make any changes to it. This is where attributes such as code documentation, understandability, and structure come into play. A poorly-documented piece of code will be harder to understand and

hence difficult to modify. Further, poorly-structured code might be harder to modify and difficult to test.

Dynamic quality attributes include software reliability, correctness, completeness, consistency, usability, and performance. Reliability refers to the probability of failure-free operation and is considered in the following section. Correctness refers to the correct operation of an application and is always with reference to some artifact. For a tester, correctness is with respect to the requirements; for a user, it is often with respect to a user manual.

Dynamic quality attributes are generally determined through multiple executions of a program. Correctness is one such attribute though one can rarely determine the correctness of a software application via testing.

Completeness refers to the availability of all features listed in the requirements, or in the user manual. An incomplete software is one that does not fully implement all features required. Of course, one usually encounters additional functionality with each new version of an application. This does not mean that a given version is incomplete because its next version has few new features. Completeness is defined with respect to a set of features that might itself be a subset of a larger set of features that are to be implemented in some future version of the application. Of course, one can easily argue that every piece of software that is correct is also complete with respect to some feature set.

Completeness refers to the availability in software of features planned and their correct implementation. Given the near impossibility of exhaustive testing, completeness is often a subjective measure.

Consistency refers to adherence to a common set of conventions and assumptions. For example, all buttons in the user interface might follow a common color coding convention. An example of inconsistency would be when a database application displays the date of birth of a person in the database; however, the date of birth is displayed in different formats, without any regard for the user's preferences, depending on which feature of the database is used.

Usability refers to the ease with which an application can be used. This is an area in itself and there exist techniques for usability testing. Psychology plays an important role in the design of techniques for usability testing. Usability testing also refers to the testing of a product by its potential users. The development organization invites a selected set of potential users and asks them to test the product. Users in turn test for ease of use, functionality as expected, performance, safety, and security. Users thus serve as an important source of tests that developers or testers within the organization might not have conceived. Usability testing is sometimes referred to as user-centric testing.

Performance refers to the time the application takes to perform a requested task. Performance is considered as a non-functional requirement. It is specified in terms such as "This task must be performed at the rate of X units of activity in one second on a machine running at speed Y, having Z gigabytes of memory." For example, the performance requirement for a compiler might be stated in terms of the minimum average time to compile of a set of numerical applications.

1.2.2 Reliability

Correctness and reliability are two dynamic attributes of software. Reliability can be considered as a statistical measure of correctness.

People want software that functions correctly every time it is used. However, this happens rarely, if ever. Most softwares that are used today contain faults that cause them to fail on some combination of inputs. Thus, the notion of total correctness of a program is an ideal and applies mostly to academic and textbook programs.

Given that most software applications are defective, one would like to know how often a given piece of software might fail. This question can be answered, often with dubious accuracy, with the help of software reliability, hereafter referred to as reliability. There are several definitions of software reliability, a few are examined below.

ANSI/IEEE STD 729-1983: RELIABILITY

Software reliability is the probability of failure-free operation of software over a given time interval and under given conditions.

The probability referred to in the definition above depends on the distribution of the inputs to the program. Such input distribution is often referred to as operational profile. According to this definition, software reliability could vary from one operational profile to another. An implication is that one user might say "this program is lousy" while another might sing praise for the same program. The following is an alternate definition of reliability.

Reliability

Software reliability is the probability of failure-free operation of software in its intended environment.

This definition is independent of "who uses what features of the software and how often." Instead, it depends exclusively on the correctness of its features. As there is no notion of operational profile, the entire input domain is considered as uniformly distributed. The term "environment" refers to the software and hardware elements needed to execute the application. These elements include the operating system, hardware requirements, and any other applications needed for communication.

Both definitions have their pros and cons. The first of the two definitions above requires the knowledge of the use profile of its users, which might be difficult or impossible to estimate accurately. However, if an operational profile can be estimated for a given class of users, then an accurate estimate of the reliability can be found for this class of users. The second definition is attractive in that one needs only one number to denote the reliability of a software application that is applicable to all its users. However, such estimates are difficult to arrive at.

1.3 Requirements, Behavior, and Correctness

Products, software in particular, are designed in response to requirements. Requirements specify the functions that a product is expected to perform. Once the product is ready, it is the requirements that determine the expected behavior. Of course, during the development of the product, the requirements might have changed from what was stated originally. Regardless of any change, the expected behavior of the product is determined by the tester's understanding of the requirements during testing.

Example 1.3 Here are the two requirements, each of which leads to a different program.

Requirement 1:	It is required to write a program that inputs two integers and outputs the maximum of these.
Requirement 2:	It is required to write a program that inputs a sequence of integers and outputs the sorted version of this sequence.

Suppose that program `max` is developed to satisfy Requirement 1 above. The expected output of `max` when the input integers are 13 and 19, can be easily determined to be 19. Now suppose that the tester wants to know if the two integers are to be input to the program on one line followed by a carriage return, or on two separate lines with a carriage return typed in after each number. The requirement as stated above fails to provide an answer to this question. This example illustrates the incompleteness of Requirement 1.

The second requirement in the above example is ambiguous. It is not clear from this requirement whether the input sequence is to be sorted in ascending or descending order. The behavior of the `sort` program, written to satisfy this requirement, will depend on the decision taken by the programmer while writing `sort`.

Testers are often faced with incomplete and/or ambiguous requirements. In such situations, a tester may resort to a variety of ways to determine what behavior to expect from the program under test. For example, forth above program `max`, one way to determine how the input should be typed in is to actually examine the program text. Another way is to ask the developer of `max` as to what decision was taken regarding the sequence in which the inputs are to be typed in. Yet another method is to execute `max` on different input sequences and determine what is acceptable to `max`.

Regardless of the nature of the requirements, testing requires the determination of the *expected* behavior of the program under test. The *observed*

behavior of the program is compared with the expected behavior to determine if the program functions as desired.

1.3.1 Input domain

A program is considered correct if it behaves as desired on all possible test inputs. Usually, the set of all possible inputs is too large for the program to be executed on each input. For example, suppose that the `max` program above is to be tested on a computer in which the integers range from −32,768 to 32,767. To test `max` on all possible integers would require it to be executed on all pairs of integers in this range. This will require a total of 2^{32} executions of `max`. It will take approximately 4.3 seconds to complete all executions assuming that testing is done on a computer that will take 1 nanosecond ($=10^{-9}$ seconds), to input a pair of integers, execute `max`, and check if the output is correct. Testing a program on all possible inputs is known as *exhaustive testing*.

According to one view, the input domain of a program consists of all possible inputs as derived from the program specification. According to another view, it is the set of all possible inputs that a program could be subjected, i.e., legal and illegal inputs.

A tester often needs to determine what constitutes "all possible inputs." The first step in determining all possible inputs is to examine the requirements. If the requirements are complete and unambiguous, it should be possible to determine the set of all possible inputs. A definition is in order before we provide an example to illustrate how to determine the set of all program inputs.

Input domain

The set of all possible inputs to a program P is known as the *input domain*, or *input space*, of P.

Example 1.4 Using Requirement 1 from Example 1.3, we find the input domain of `max` to be the set of all pairs of integers where each element in the pair is in the range −32,768 till 32,767.

Example 1.5 Using Requirement 2 from Example 1.3, it is not possible to find the input domain for the `sort` program. Let us therefore assume that the requirement was modified to be the following:

Modified Requirement 2:	It is required to write a program that inputs a sequence of integers and outputs the integers in this sequence sorted in either ascending or descending order. The order of the output sequence is determined by an input request

character that should be "A" when an ascending sequence is desired, and "D" otherwise. While providing input to the program, the request character is entered first followed by the sequence of integers to be sorted; the sequence is terminated with a period.

Based on the above modified requirement, the input domain for `sort` is a set of pairs. The first element of the pair is a character. The second element of the pair is a sequence of zero or more integers ending with a period. For example, following are three elements in the input domain of `sort`:

```
< A -3 15 12 55.>
< D 23 78.>
< A .>
```

The first element contains a sequence of four integers to be sorted in ascending order, the second element has a sequence to be sorted in descending order, and the third element has an empty sequence to be sorted in ascending order.

We are now ready to give the definition of program correctness.

Correctness

A program is considered correct if it behaves as expected on each element of its input domain.

1.3.2 Specifying program behavior

There are several ways to define and specify program behavior. The simplest way is to specify the behavior in a natural language such as English. However, this is more likely subject to multiple interpretations than a more formally specified behavior. Here, we explain how the notion of program "state" can be used to define program behavior and how the "state transition diagram," or simply "state diagram," can be used to specify program behavior.

A collection of the current values of program variables and the location of control, is considered as a state vector for that program.

The "state" of a program is the set of current values of all its variables and an indication of which statement in the program is to be executed next. One way to encode the state is by collecting the current values of program variables into a vector known as the "state vector." An indication of where

the control of execution is at any instant of time can be given by using an identifier associated with the next program statement. In the case of programs in assembly language, the location of control can be specified more precisely by giving the value of the program counter.

Each variable in the program corresponds to one element of this vector. Obviously, for a large program, such as the Unix operating system, the state vector might have thousands of elements. Execution of program statements causes the program to move from one state to the next. A sequence of program states is termed as program behavior.

Example 1.6 Consider a program that inputs two integers into variables X and Y, compares these values, sets the value of Z to the larger of the two, displays the value of Z on the screen, and exits. Program P1.1 shows the program skeleton. The state vector for this program consists of four elements. The first element is the statement identifier where the execution control is currently at. The next three elements are, respectively, the values of the three variables X, Y, and Z.

Program P1.1

```
1  integer X, Y, Z;
2  input (X, Y);
3  if (X < Y)
4    {Z=Y;}
5  else
6    {Z=X;}
7  endif
8  output (Z);
9  end
```

The letter `u` as an element of the state vector stands for an "undefined" value. The notation $s_i \rightarrow s_j$ is an abbreviation for "The program moves from state s_i to s_j." The movement from s_i to s_j is caused by the execution of the statement whose identifier is listed as the first element of state s_i. A possible sequence of states that the `max` program may go through is given below.

$$[2\ \mathtt{u}\ \mathtt{u}\ \mathtt{u}] \rightarrow [3\ \mathtt{3}\ \mathtt{15}\ \mathtt{u}] \rightarrow [4\ \mathtt{3}\ \mathtt{15}\ \mathtt{15}] \rightarrow [5\ \mathtt{3}\ \mathtt{15}\ \mathtt{15}]$$
$$\rightarrow [8\ \mathtt{3}\ \mathtt{15}\ \mathtt{15}] \rightarrow [9\ \mathtt{3}\ \mathtt{15}\ \mathtt{15}]$$

Upon the start of its execution, a program is in an "initial state." A (correct) program terminates in its "final state." All other program states are termed as "intermediate states." In Example 1.6, the initial state is

[2 u u u], the final state is [9 3 15 15], and there are four intermediate states as indicated.

Program behavior can be modeled as a sequence of states. With every program one can associate one or more states that need to be observed to decide whether or not the program behaves according to its requirements. In some applications it is only the final state that is of interest to the tester. In other applications a sequence of states might be of interest. More complex patterns might also be needed.

A sequence of states is representative of program behavior.

Example 1.7 For the `max` program (P1.1), the final state is sufficient to decide if the program has successfully determined the maximum of two integers. If the numbers input to `max` are 3 and 15, then the correct final state is [9 3 15 15]. In fact, it is only the last element of the state vector, `15`, which may be of interest to the tester.

Example 1.8 Consider a menu-driven application named `myapp`. Figure 1.2 shows the menu bar for this application. It allows a user to position and click the mouse on any one of a list of menu items displayed in the menu bar on the screen. This results in the "pulling down" of the menu and a list of options is displayed on the screen. One of the items on the menu bar is labeled `File`. When `File` is pulled down, it displays `Open` as one of several options. When the `Open` option is selected, by moving the cursor over it, it should be

Figure 1.2 Menu bar displaying four menu items when application `myapp` is started.

Figure 1.3 A state sequence for `myapp` showing how the application is expected to behave when the user selects the `open` option under the `File` menu item.

highlighted. When the mouse is released, indicating that the selection is complete, a window displaying names of files in the current directory should be displayed.

Figure 1.3 depicts the sequence of states that `myapp` is expected to enter when the user actions described above are performed. When started, the application enters the initial state wherein it displays the menu bar and waits for the user to select a menu item. This state diagram depicts the expected behavior of `myapp` in terms of a state sequence. As shown in Figure 1.3, `myapp` moves from state s_0 to state s_3 after the sequence of actions t_0, t_1, t_2, and t_3 has been applied. To test `myapp`, the tester could apply the sequence of actions depicted in this state diagram and observe if the application enters the expected states.

As you might observe from Figure 1.3, a state sequence diagram can be used to specify the *behavioral* requirements of an application. This same specification can then be used during testing to ensure if the application conforms to the requirements.

1.3.3 Valid and invalid inputs

A program ought to be tested based on representatives from the set of valid as well as invalid inputs. The latter set is used to determine the robustness of a program.

In the examples above, the input domains are derived from the requirements. However, due to the incompleteness of requirements, one might have to think a bit harder to determine the input domain. To illustrate why, consider the modified requirement in Example 1.5. The requirement mentions that the request characters can be "A" or "D", but it fails to answer the question "What if the user types a different character?" When using `sort` it is certainly possible for the user to type a character other than "A" or "D". Any character other than "A" or "D" is considered as invalid input to `sort`. The requirement for `sort` does not specify what action it should take when an invalid input is encountered.

Identifying the set of invalid inputs and testing, the program against these inputs is an important part of the testing activity. Even when the requirements fail to specify the program behavior on invalid inputs, the programmer does treat these in one way or another. Testing a program against invalid inputs might reveal errors in the program.

Example 1.9 Suppose that we are testing the `sort` program. We execute it against the following input: $< E\ 7\ 19\ .\ >$. The requirements in Example 1.5 are insufficient to determine the expected behavior of `sort` on the above input. Now suppose that upon execution on the above input, the `sort` program enters into an infinite loop and neither asks the user for any input nor responds to anything typed by the user. This observed behavior points to a possible error in `sort`.

The argument above can be extended to apply to the sequence of integers to be sorted. The requirements for the `sort` program do not specify how the program should behave if, instead of typing an integer, a user types in a character, such as "?". Of course, one would say, the program should inform the user that the input is invalid. But this expected behavior from `sort` needs to be tested. This suggests that the input domain for `sort` should be modified.

Example 1.10 Considering that `sort` may receive valid and invalid inputs, the input domain derived in Example 1.5 needs modification. The modified input domain consists of pairs of values. The first value in the pair is any ASCII character that can be typed by a user as a request character. The second element of the pair is a sequence of integers, interspersed with invalid characters,

terminated by a period. Thus, for example, the following are sample elements from the modified input domain:

```
< A 7 19.>
< D 7 9F 19.>
```

In the example above, we assumed that invalid characters are possible inputs to the `sort` program. This, however, may not be the case in all situations. For example, it might be possible to guarantee that the inputs to `sort` will always be correct as determined from the modified requirements in Example 1.5. In such a situation, the input domain need not be augmented to account for invalid inputs if the guarantee is to be believed by the tester.

In cases where the input to a program is not guaranteed to be correct, it is convenient to partition the input domain into two subdomains. One subdomain consists of inputs that are valid and the other consists of inputs that are invalid. A tester can then test the program on selected inputs from each subdomain.

1.4 Correctness Versus Reliability

1.4.1 Correctness

Program correctness is established via mathematical proofs. Program reliability is established via testing. However, neither approach is foolproof.

Though correctness of a program is desirable, it is almost never the objective of testing. To establish correctness via testing would imply testing a program on all elements in the input domain. In most cases that are encountered in practice, this is impossible to accomplish. Thus, correctness is established via mathematical proofs of programs. A proof uses the formal specification of requirements and the program text to prove or disprove that the program will behave as intended. While a mathematical proof is precise, it too is subject to human errors. Even when the proof is carried out by a computer, the simplification of requirements specification and errors in tasks that are not fully automated might render the proof useless.

Testing a program for completeness does not ensure its correctness.

While correctness attempts to establish that the program is error free, testing attempts to find if there are any errors in it. Completeness of testing does not necessarily demonstrate that a program is error free. However, as testing progresses, errors might be revealed. Removal of errors from the program usually improves the chances, or the probability, of the program executing without any failure. Also, testing, debugging, and the error removal processes together increase our confidence in the correct functioning of the program under test.

Example 1.11 This example illustrates why the probability of program failure might not change upon error removal. Consider the following program that inputs two integers x and y and prints the value of $f(x, y)$ or $g(x, y)$ depending on the condition $x < y$.

```
integer x, y
input x, y
if (x < y)        ← This condition should be x ≤ y.
  {print f(x,y)}
else
  {print g(x,y)}
```

The above program uses two functions f and g not defined here. Let us suppose that function f produces an incorrect result whenever it is invoked with $x = y$ and that $f(x, y) \neq g(x, y), x = y$. In its present form, the program fails when tested with equal input values because function g is invoked instead of function f. When the error is removed by changing the condition $x < y$ to $x \leq y$, the program fails again when the input values are the same. The latter failure is due to the error in function f. In this program, when the error in f is also removed, the program will be correct assuming that all other code is correct.

Reliability is a statistical attribute. According to one view, it is the probability of failure free execution of a program under given conditions.

1.4.2 Reliability

The probability of a program failure is captured more formally in the term "reliability." Consider the second of the two definitions examined earlier: "The *reliability* of a program P is the probability of its successful execution on a randomly selected element from its input domain."

A comparison of program correctness and reliability reveals that while correctness is a binary metric, reliability is a continuous metric over a scale from 0 to 1. A program can be either correct or incorrect; its reliability can be anywhere between 0 and 1. Intuitively, when an error is removed from a program, the reliability of the program so obtained is expected to be higher than that of the one that contains the error. As illustrated in the example above, this may not be always true. The next example illustrates how to compute program reliability in a simplistic manner.

Example 1.12 Consider a program P that takes a pair of integers as input. The input domain of this program is the set of all pairs of integers. Suppose now that in actual use there are only three pairs that will be input to P. These are

$$\{<(0,0)\ (-1,1)\ (1,-1)>\}$$

The above set of three pairs is a subset of the input domain of P and is derived from a knowledge of the actual use of P, and not solely from its requirements.

Suppose also that each pair in the above set is equally likely to occur in practice. If it is known that P fails on exactly one of the three possible input pairs then the frequency with which P will function correctly is $\frac{2}{3}$. This number is an estimate of the probability of the successful operation of P and hence is the reliability of P.

1.4.3 Operational profiles

An operational profile is a statistical summary of how a program would be used in practice.

As per the definition above, the reliability of a program depends on how it is used. Thus, in Example 1.12, if P is never executed on input pair $(0, 0)$, then the restricted input domain becomes $\{< (-1, 1)\ (1, -1) >\}$ and the reliability of P is 1. This leads us to the definition of *operational profile*.

Operational profile

An operational profile is a numerical description of how a program is used.

In accordance with the above definition, a program might have several operational profiles depending on its users.

Example 1.13 Consider a `sort` program that, on any given execution, allows any one of two types of input sequences. One sequence consists of numbers only and the other consists of alphanumeric strings. One operational profile for `sort` is specified as follows:

Operational profile #1	
Sequence	Probability
Numbers only	0.9
Alphanumeric strings	0.1

Another operational profile for `sort` is specified as follows:

Operational profile #2	
Sequence	Probability
Numbers only	0.1
Alphanumeric strings	0.9

The two operational profiles above suggest significantly different uses of `sort`. In one case, it is used mostly for sorting sequences

of numbers, and in the other case, it is used mostly for sorting alphanumeric strings.

1.5 Testing and Debugging

Testing is the process of determining if a program behaves as expected. In the process, one may discover errors in the program under test. However, when testing reveals an error, the process used to determine the cause of this error and to remove it is known as *debugging*. As illustrated in Figure 1.4, testing and debugging are often used as two related activities in a cyclic manner.

Testing and debugging are two distinct though intertwined activities. Testing generally leads to debugging though both activities might not be always performed by the same individual.

1.5.1 Preparing a test plan

A test cycle is often guided by a *test plan*. When relatively small programs are being tested, a test plan is usually informal and in the tester's mind, or there may be no plan at all. A sample test plan for testing the `sort` program is shown in Figure 1.5.

The sample test plan in Figure 1.5 is often augmented by items such as the method used for testing, method for evaluating the adequacy of test cases, and method to determine if a program has failed or not.

1.5.2 Constructing test data

A test case is a pair of input data and the corresponding program output; a test set is a collection of test cases.

A *test case* is a pair of input data and the corresponding program output. The test data are a set of values: one for each input variable. A *test set* is a collection of test cases. A test set is sometimes referred to as a test suite. The notion of "one execution" of a program is rather tricky and is elaborated later in this chapter. *Test data* is an alternate term for test set.

Program requirements and the test plan help in the construction of test data. Execution of the program on test data might begin after all or a few test cases have been constructed. While testing, relatively small programs testers often generate a few test cases and execute the program against these. Based on the results obtained, the tester decides whether to continue the construction of additional test cases or to enter the debugging phase.

Example 1.14 The following test cases are generated for the `sort` program using the test plan in Figure 1.5.

```
Test case 1:
  Test data:       <"A" 12 -29 32.>
```

Figure 1.4 A test and debug cycle.

```
  Expected output: -29 12 32

Test case 2:
  Test data:       <"D" 12 -29 32.>
  Expected output: 32 12 -29

Test case 3:
  Test data:       <"A".>
  Expected output: No input to be sorted in ascending
                   order.

Test case 4:
  Test data:       <"D".>
```

Test Plan for `sort`.

The `sort` program is to be tested to meet the requirements given in Example 1.5. Specifically, the following needs to be done:

1. Execute the program on at least two input sequences, one with "A" and the other with "D" as request characters.
2. Execute the program on an empty input sequence.
3. Test the program for robustness against erroneous inputs such as "R" typed in as the request character.
4. All failures of the test program should be recorded in a suitable file using the Company Failure Report Form.

Figure 1.5 A sample test plan for the `sort` program.

```
    Expected output: No input to be sorted in ascending
                     order.

  Test case 5:
    Test data:        <"R" 3 17.>
    Expected output: Invalid request character
    Valid characters: "A" and "D".

  Test case 6:
    Test data:       <"A" c 17.>
    Expected output: Invalid number.
```

Test cases 1 and 2 are derived in response to item 1 in the test plan; 3 and 4 are in response to item 2. Notice that we have designed two test cases in response to item 2 of the test plan even though the plan calls for only 1 test case. Notice also that the requirements for the `sort` program as in Example 1.5 do not indicate what should be the output of `sort` when there is nothing to be sorted. We therefore took an arbitrary decision while composing the "Expected output" for an input that has no numbers to be sorted. Test cases 5 and 6 are in response to item 3 in the test plan.

As is evident from the above example, one can select a variety of test sets to satisfy the test plan requirements. Questions such as "Which test set is the best?" and "Is a given test set adequate?" are answered in Part III of this book.

1.5.3 Executing the program

Execution of a program under test is the next significant step in testing. Execution of this step for the `sort` program is most likely a trivial exercise. However, this may not be so for large and complex programs. For example,

A test harness is an aid to testing a program.

to execute a program that controls a digital cross connect switch used in telephone networks, one may first need to follow an elaborate procedure to load the program into the switch and then yet another procedure to input the test cases to the program. Obviously, the complexity of actual program execution is dependent on the program itself.

Often a tester might be able to construct a *test harness* to aid in program execution. The harness initializes any global variables, inputs a test case, and executes the program. The output generated by the program may be saved in a file for subsequent examination by a tester. The next example illustrates a simple test harness.

Example 1.15 The test harness in Figure 1.6 reads an input sequence, checks for its correctness, and then calls `sort`. The sorted array `sorted_sequence` returned by `sort` is printed using the `print_sequence` procedure. Test cases are assumed to be available in the Test pool file shown in the figure. In some cases, the tests might be generated from within the harness.

In preparing this test harness we assume that (a) `sort` is coded as a procedure, (b) the `get_input` procedure reads the request character and the sequence to be sorted into variables `request_char`, `num_items`, and `in_numbers`, and (c) the input is checked prior to calling `sort` by the `check_input` procedure.

The `test_setup` procedure is usually invoked first to set up the test that, in this example, includes identifying and opening the file containing tests. The `check_output` procedure serves as

Figure 1.6 A simple test harness to test the `sort` program.

the oracle that checks if the program under test behaves correctly. The `report_failure` procedure is invoked in case the output from `sort` is incorrect. A failure might be simply reported via a message on the screen or saved in a test report file (not shown in Figure 1.6). The `print_sequence` procedure prints the sequence generated by the `sort` program. The output generated by `print_sequence` can also be piped into a file for subsequent examination.

1.5.4 Assessing program correctness

An important step in testing a program is the one wherein the tester determines if the observed behavior of the program under test is correct or not. This step can be further divided into two smaller steps. In the first step one observes the behavior and in the second step analyses the observed behavior to check if it is correct or not. Both these steps can be trivial for small programs, such as for `max` in Example 1.3, or extremely complex as in the case of a large distributed software system. The entity that performs the task of checking the correctness of the observed behavior is known as an *oracle*. Figure 1.7 shows the relationship between the program under test and the oracle.

Figure 1.7 Relationship between the program under test and the oracle. The output from an oracle can be binary such as "Yes" or "No" or more complex such as an explanation as to why the oracle finds the observed behavior to be same or different from the expected behavior.

An oracle is something that checks the behavior of a program. An oracle could itself be a program or a human being.

A tester often assumes the role of an oracle and thus serves as a "human oracle." For example, to verify if the output of a matrix multiplication program is correct or not, a tester might input two 2×2 matrices and check if the output produced by the program matches the results of hand calculation. As another example, consider checking the output of a text processing program. In this case, a human oracle might visually inspect the monitor screen to verify whether or not the italicize command works correctly when applied to a block of text.

Checking program behavior by humans has several disadvantages. First, it is error prone as the human oracle might make an error in analysis. Second, it may be slower than the speed with which the program computed the results. Third, it might result in the checking of only trivial input–output behaviors. However, regardless of these disadvantages, a human oracle can often be the best available oracle.

Oracles can also be programs designed to check the behavior of other programs. For example, one might use a matrix multiplication program to check if a matrix inversion program has produced the correct output. In this case, the matrix inversion program inverts a given matrix A and generates B as the output matrix. The multiplication program can check to see if $A \times B = I$ within some bounds on the elements of the identity matrix I. Another example is an oracle that checks the validity of the output from a sort program. Assuming that the sort program sorts input numbers in ascending order, the oracle needs to check if the output of the sort program is indeed in ascending order.

Using programs as oracles has the advantage of speed, accuracy, and the ease with which complex computations can be checked. Thus, a matrix multiplication program, when used as an oracle for a matrix inversion program, can be faster, accurate, and check very large matrices when compared to the same function performed by a human oracle.

1.5.5 Constructing an oracle

Construction of an automated oracle, such as the one to check a matrix multiplication or a sort program, requires the determination of input–output relationship. For matrix inversion, and for sorting, this relation is rather simple and expressed precisely in terms of a mathematical formula or a simple algorithm as explained above. Also, when tests are generated from models such as finite state machines or statecharts, both inputs and the corresponding outputs are available. This makes it possible to construct an oracle while generating the tests. However, in general, the construction of an automated oracle is a complex undertaking. The next example illustrates one method for constructing an oracle.

Example 1.16 Consider a program named `HVideo` that allows one to keep track of old home videos. The program operates in two modes: data entry and search. In the date entry mode, it displays a screen into which the user types in information about a video cassette. This information includes data such as the title, the comments, and the date when the video was prepared. Once the information is typed the user clicks on the `Enter` button, which results in this information being added to a database. In the search mode, the program displays a screen into which a user can type some attribute of the video being searched for and sets up a search criterion. A sample criterion is "Find all videos that contain the name `Sonia` in the title field." In response, the program returns all videos in the database that match the search criteria or displays an appropriate message if no match is found.

To test `HVideo`, we need to create an oracle that checks whether or not the program functions correctly in data entry and search modes. In addition, an input generator needs to be created. As shown in Figure 1.8, the input generator generates inputs for `HVideo`. To test the data entry operation of `HVideo`, the input generator generates a data entry request. This request consists of an operation code, `Data Entry`, and the data to be entered that includes the title, the comments, and the date on which the video was prepared. Upon completion of execution of the `Enter` request, `HVideo` returns control to the input generator. The input generator now requests the oracle to test if `HVideo` performed its task correctly on the input given for data entry. The oracle uses the input to check if the information to be entered into the database has been entered correctly or not. The oracle returns a Pass or No Pass to the input generator.

To test if `HVideo` correctly performs the search operation, the input generator formulates a search request with the search data the same as the one given previously with the `Enter` command. This

Figure 1.8 Relationship between an input generator, `HVideo`, and its oracle.

input is passed on to `HVideo` that performs the search and returns the results of the search to the input generator. The input generator passes these results to the oracle to check for their correctness. There are at least two ways in which the oracle can check for the correctness of the search results. One is for the oracle to actually search the database. If its findings are the same as that of `HVideo`, then the search results are assumed to be correct and incorrect otherwise.

A test metric measures some aspect of the test process. Test metrics could be at various levels such as at the level of an organization, a project, a process or a product.

1.6 Test Metrics

The term "metric" refers to a standard of measurement. In software testing, there exist a variety of metrics. Figure 1.9 shows a classification of various types of metrics briefly discussed in this section. Metrics can be computed at the organizational, process, project, and product levels. Each set of measurements has its value in monitoring, planning, and control.

Regardless of the level at which metrics are defined and collected, there exist four general core areas that assist in the design of metrics. These are schedule, quality, resources, and size. Schedule-related metrics measure actual completion times of various activities and compare these with estimated time to completion. Quality-related metrics measure the quality of a product or a process. Resource-related metrics measure items such as cost in dollars, manpower, and tests executed. Size-related metrics measure the size of various objects such as the source code and the number of tests in a test suite.

1.6.1 Organizational metrics

Metrics at the level of an organization are useful in overall project planning and management. Some of these metrics are obtained by aggregating compatible metrics across multiple projects. Thus, for example, the number

Figure 1.9 Types of metrics used in software testing and their relationships.

of defects reported after product release, averaged over a set of products developed and marketed by an organization, is a useful metric of product quality at the organizational level.

Computing this metric at regular intervals and overall products released over a given duration shows the quality trend across the organization. For example, one might say "The number of defects reported in the field over all products and within three months of their shipping, has dropped from 0.2 defects per KLOC (thousand lines of code) to 0.04 defects KLOC." Other organizational level metrics include testing cost per KLOC, delivery schedule slippage, and time to complete system testing.

Defect density is the number of defects per line of code. Defect density is a widely used metric at the level of a project.

Organizational level metrics allow senior management to monitor the overall strength of the organization and points to areas of weaknesses. These metrics help the senior management in setting new goals and plan for resources needed to realize these goals.

Example 1.17 The average defect density across all software projects in a company is 1.73 defects per KLOC. Senior management has found that for the next generation of software products, which they plan to bid, they need to show that product density can be reduced to 0.1 defects per KLOC. The management thus sets a new goal.

Given the time frame from now until the time to bid, the management needs to do a feasibility analysis to determine whether or not this goal can be met. If a preliminary analysis shows that it can be met, then a detailed plan needs to be worked out and put into action. For example, management might decide to train its employees in the use of new tools and techniques for defect prevention and detection using sophisticated static analysis techniques.

1.6.2 Project metrics

Project metrics relate to a specific project, for example, the I/O device testing project or a compiler project. These are useful in the monitoring and control of a specific project. The ratio of actual to planned system test effort is one project metric. Test effort could be measured in terms of the tester-man-months. At the start of the system test phase, for example, the project manager estimates the total system test effort. The ratio of actual to estimated effort is zero prior to the system test phase. This ratio builds up over time. Tracking the ratio assists the project manager in allocating testing resources.

Another project metric is the ratio of the number of successful tests to the total number of tests in the system test phase. At any time during the

project, the evolution of this ratio from the start of the project could be used to estimate the time remaining to complete the system test process.

1.6.3 Process metrics

Every project uses some test process. The "big bang" approach is one process sometimes used in relatively small single person projects. Several other well-organized processes exist. The goal of a process metric is to assess the "goodness" of the process.

The purpose of a process metric is to assess the "goodness" of a process.

When a test process consists of several phases, for example, unit test, integration test, system test, etc, one can measure how many defects were found in each phase. It is well known that the later a defect is found, the costlier it is to fix. Hence, a metric that classifies defects according to the phase in which they are found assists in evaluating the process itself.

Example 1.18 In one software development project, it was found that 15% of the total defects were reported by customers, 55% of the defects prior to shipping were found during system test, 22% during integration test, and the remaining during unit test. The large number of defects found during the system test phase indicates a possibly weak integration and unit test process. The management might also want to reduce the fraction of defects reported by customers.

1.6.4 Product metrics: generic

Product metrics relate to a specific product such as a compiler for a programming language. These are useful in making decisions related to the product, for example, "Should this product be released for use by the customer?"

Product quality-related metrics abound. We introduce two types of metrics here: the cyclomatic complexity and the Halstead metrics. The cyclomatic complexity proposed by Thomas McCabe in 1976 is based on the control flow of a program. Given the CFG G (see Chapter 2.2 for details) of program P containing N nodes, E edges, and p connected procedures, the cyclomatic complexity $V(G)$ is computed as follows:

The cyclomatic complexity is a measure of program complexity; higher values imply higher complexity. This is a product metric.

$$V(G) = E - N + 2p$$

Note that P might consist of more than one procedure. The term p in $V(G)$ counts only procedures that are reachable from the main function. $V(G)$ is the complexity of a CFG G that corresponds to a procedure reachable

from the main procedure. Also, $V(G)$ is not the complexity of the entire program, instead it is the complexity of a procedure in P that corresponds to G (see Exercise 1.12). Larger values of $V(G)$ tend to imply higher program complexity and hence a program more difficult to understand and test than one with a smaller values. $V(G)$ is 5 or less are recommended.

The now well-known Halstead complexity measures were published by late Professor Maurice Halstead in a book titled "Elements of Software Science." Table 1.2 lists some of the software science metrics. Using program size (S) and effort (E), the following estimator has been proposed for the number of errors (B) found during a software development effort:

$$B = 7.6E^{0.667}S^{0.333}$$

Extensive empirical studies have been reported to validate Halstead's software science metrics. An advantage of using an estimator such as B is that it allows the management to plan for testing resources. For example, a larger value of the number of expected errors will lead to a larger number of testers and testing resources to complete the test process over a given duration. Nevertheless, modern programming languages such as Java and C++ do not lend themselves well to the application of the software science metrics. Instead, one uses specially devised metrics for object-oriented languages described next (also see Exercise 1.16).

Table 1.2 Halstead measures of program complexity and effort.

Measure	Notation	Definition
Operator count	N_1	Number of operators in a program
Operand count	N_2	Number of operands in a program.
Unique operators	η_1	Number of unique operators in a program
Unique operands	η_2	Number of unique operands in a program
Program vocabulary	η	$\eta_1 + \eta_2$
Program size	N	$N_1 + N_2$
Program volume	V	$N \times \log_2 \eta$
Difficulty	D	$2/\eta_1 \times \eta_2/N_2$
Effort	E	$D \times V$

1.6.5 Product metrics: OO software

A number of empirical studies have investigated the correlation between product complexity metric application quality. Table 1.3 lists a sample of product metrics for object-oriented and other applications. Product reliability is a quality metric and refers to the probability of product failure for a given operational profile. As explained in Section 1.4.2, product reliability of software truly measures the probability of generating a failure causing test input. If for a given operational profile and in a given environment this probability is 0, then the program is perfectly reliable despite the possible presence of errors. Certainly, one could define other metrics to assess software reliability. A number of other product quality metrics, based on defects, are listed in Table 1.3.

The OO metrics in the table are due to Shyam Chidamber and Chris Kemerer. They measure program or design complexity. They are of direct relevance to testing in that a product with a complex design will likely require more test effort to obtain a given level of defect density than a product with less complexity.

Table 1.3 A sample of product metrics.

Metric	Meaning
Reliability	Probability of failure of a software product with respect to a given operational profile in a given environment.
Defects density	Number of defects per KLOC.
Defect severity	Distribution of defects by their level of severity
Test coverage	Fraction of testable items, e.g., basic blocks, covered. Also, a metric for test adequacy or "goodness of tests."
Cyclomatic complexity	Measures complexity of a program based on its CFG.
Weighted methods per class	$\sum_{i=1}^{n} c_i$, where c_i is the complexity of method i in the class under consideration.
Class coupling	Measures the number of classes to which a given class is coupled.
Response set	Set of all methods that can be invoked, directly and indirectly, when a message is sent to object O.
Number of children	Number of immediate descendants of a class in the class hierarchy.

1.6.6 Progress monitoring and trends

Metrics are often used for monitoring progress. This requires making measurements on a regular basis over time. Such measurements offer trends. For example, suppose that a browser has been coded, unit tested, and its components integrated. It is now in the system testing phase. One could measure the cumulative number of defects found and plot these over time. Such a plot will rise over time. Eventually, it will likely show a saturation indicating that the product is reaching a stability stage. Figure 1.10 shows a sample plot of new defects found over time.

Product metrics could be classified as static or dynamic. Computing a dynamic metric will likely require program execution.

1.6.7 Static and dynamic metrics

Static metrics are those computed without having to execute the product. Number of testable entities in an application is an example of a static product metric. Dynamic metrics require code execution. For example, the number of testable entities actually covered by a test suite is a dynamic product metric.

One could apply the notions of static and dynamic metrics to organization and project. For example, the average number of testers working on a project is a static project metric. Number of defects remaining to be fixed could be treated as a dynamic metric as it can be computed accurately only after a code change has been made and the product retested.

1.6.8 Testability

According to IEEE, testability is the "degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met." Different ways to measure testability of a product can be categorized into static and dynamic testability metrics. Software complexity is one static testability metric. The more complex an application, the lower the testability, that is, the higher the effort required to test it. Dynamic metrics for testability

Figure 1.10 A sample plot of cumulative count of defects found over seven consecutive months in a software project.

include various code-based coverage criteria. For example, a program for which it is difficult to generate tests that satisfy the statement coverage criterion is considered to have low testability than one for which it is easier to construct such tests.

The testability of a program is the degree to which a program facilitates the establishment and measurement of some performance criteria.

High testability is a desirable goal. This is done best by knowing what needs to be tested and how, well in advance. It is recommended that features to be tested and how they are to be tested must be identified during the requirements stage. This information is then updated during the design phase and carried over to the coding phase. A testability requirement might be met through the addition of some code to a class. In more complex situations, it might require special hardware and probes in addition to special features aimed solely at meeting a testability requirement.

Example 1.19 Consider an application E required to control the operation of an elevator. E must pass a variety of tests. It must also allow a tester to perform a variety of tests. One test checks if the elevator hoists motor and the brakes are working correctly. Certainly one could do this test when the hardware is available. However, for concurrent hardware and software development, one needs a simulator for the hoist motor and brake system.

To improve the testability of E, one must include a component that allows it to communicate with a hoist motor and brake simulator and display the status of the simulated hardware devices. This component must also allow a tester to input tests such as "start the motor."

Another test requirement for E is that it must allow a tester to experiment with various scheduling algorithms. This requirement can be met by adding a component to E that offers a palette of scheduling algorithms to choose from and whether or not they have been implemented. The tester selects an implemented algorithm and views the elevator movement in response to different requests. Such testing also requires a "random request generator" and a display of such requests and the elevator response (also see Exercise 1.14).

Testability is a concern in both hardware and software designs. In hardware design, testability implies that there exist tests to detect any fault with respect to a fault model in a finished product. Thus, the aim is to verify the correctness of a finished product. Testability in software focuses on the verification of design and implementation.

1.7 Software and Hardware Testing

There are several similarities and differences between techniques used for testing software and hardware. It is obvious that a software application does not degrade over time, any fault present in the application will remain, and no new faults will creep in unless the application is changed. This is not true for hardware, such as a VLSI chip, that might fail over time due to a fault that did not exist at the time the chip was manufactured and tested.

This difference in the development of faults during manufacturing or over time leads to built-in self test (BIST) techniques applied to hardware designs and rarely, if at all, to software designs and code. BIST can be applied to software but will only detect faults that were present when the last change was made. Note that internal monitoring mechanisms often installed in software are different from BIST intended to actually test or the correct functioning of a circuit.

Fault models: Hardware testers generate tests based on fault models. For example, using a *stuck-at* fault model a set of input test patterns can be used to test whether or not a logic gate is functioning as expected. The fault being tested for is a manufacturing flaw or might have developed due to degradation over time. Software testers generate tests to test for correct functionality. Sometimes such tests do not correspond to any general fault model. For example, to test whether or not there is a memory leak in an application, one performs a combination of stress testing and code inspection. A variety of faults could lead to memory leaks.

Hardware testers use a variety of fault models at different levels of abstraction. For example, at the lower level there are transistor level faults. At higher levels there are gate level, circuit level, and function level fault models. Software testers might or might not use fault models during test generation even though the models exist. Mutation testing described in Chapter 8 is a technique based on software fault models. Other techniques for test generation such as condition testing, finite state model-based testing, and combinatorial designs are also based on well-defined fault models which shall be discussed in Chapters 4–6, respectively. Techniques for automatic generation of tests, as described in several chapters in Part II of this book are based on precise fault models.

Test domain: A major difference between tests for hardware and software is in the domain of tests. Tests for a VLSI chip, for example, take the form of a bit pattern. For combinational circuits, for example, a multiplexer, a finite set of bit patterns will ensure the detection of any fault with respect to a circuit level fault model. For sequential circuits that use flip-flops, a test may be a sequence of bit patterns that move the circuit from one state to another and a test suite is a collection of such tests. For software, the

Test domains differ between hardware and software tests. However, mutation testing of software is motivated by concepts in hardware testing.

domain of a test input is different than that for hardware. Even for the simplest of programs, the domain could be an infinite set of tuples with each tuple consisting of one or more basic data types such as integers and reals.

Stuck-at fault models are common in hardware.

Example 1.20 Consider a simple 2-input NAND gate in Figure 1.11(a). The stuck-at fault model defines several faults at the input and output of a logic gate. Figure 1.11(b) shows a 2-input NAND gate with a stuck-at-1 fault, abbreviated as *s-a-1*, at input A. The truth tables for the correct and the faulty NAND gates are shown below.

Correct NAND gate			Faulty NAND gate		
A	B	O	A	B	O
0	0	1	0(1)	0	1
0	1	1	0(1)	1	0
1	0	1	1(1)	0	1
1	1	0	1(1)	1	0

A test bit vector $v : (A = 0, B = 1)$ leads to output 0 whereas the correct output should be 1. Thus, v detects a single s-a-1 fault in the A input of the NAND gate. There could be multiple stuck-at faults. Exercise 1.15 asks you to determine whether or not multiple stuck-at faults in a 2-input NAND gate can always be detected.

Test coverage: It is practically impossible to completely test a large piece of software, for example, an operating system as well as a complex integrated circuit such as a modern 32 or 64-bit microprocessor. This leads to the notion of acceptable test coverage. In VLSI testing, such coverage is measured using a fraction of the faults covered to the total that might be present with respect to a given fault model.

The idea of fault coverage in hardware is also used in software testing using program mutation. A program is mutated by injecting a number of faults using a fault model that corresponds to mutation operators. The

Figure 1.11 (a) A 2-input NAND gate. (b) A NAND gate with a stuck-at 1 fault in input A.

adequacy of a test set is assessed as a fraction of the mutants covered to the total number mutants. Details of this technique are described in Chapter 8.

Test set efficiency: There are multiple definitions of test set efficiency. Here we give one. Let T be a test set developed to test program P that must meet requirements R. Let f be the number of faults in P and f' the number faults detected when P is executed against test cases in T. The efficiency of T is measured as the ratio $\frac{f'}{f}$. Note that the efficiency is a number between 0 and 1. This definition of test set efficiency is also referred to as the *effectiveness* of a test set.

In most practical situations, one would not know the number of faults in a program under test. Hence, in such situations it is not possible to measure precisely the efficiency of a test set. Nevertheless, empirical studies are generally used to measure the efficiency of a test set by using well-known and understandable programs that are seeded with known faults.

Note that the size of a test set is not accounted for in the above definition of efficiency. However, given two test sets with equal efficiency, it is easy to argue that one would prefer to use the one that contains fewer tests.

The term *test efficiency* is different from *test set* or *test suite* efficiency. While the latter refers to the efficiency of a test suite, test efficiency refers to the efficiency of the test process. The efficiency of a test process is often defined as follows: Let D denote the total number of defects found during the various phases of testing excluding the user acceptance phase. Let D_u be the number of defects found in user acceptance testing and $D' = D + D_u$. Then, test efficiency is defined as $\frac{D}{D'}$.

Notice that test suite efficiency and test efficiency are two quite different metrics. Test set efficiency is an indicator of the goodness of the techniques used to develop the tests. Test efficiency is an indicator of the goodness of the entire test process, and hence it includes in some ways, test set metrics.

1.8 Testing and Verification

Testing aims to find if there are any errors in the program. It does so by sampling the input domain and checking the program behavior against the samples. Verification aims at proving the correctness of a program or a specific part of it.

Program verification aims at proving the correctness of programs by showing that it contains no errors. This is very different from testing that aims at uncovering errors in a program. While verification aims at showing that a given program works for all possible inputs that satisfy a set of conditions, testing aims to show that the given program is reliable in that no errors of any significance were found.

Program verification and testing are best considered as complementary techniques. In practice, one often sheds program verification, but not testing. However, in the development of critical applications, such as smart

cards, or control of nuclear plants, one often makes use of verification techniques to prove the correctness of some artifact created during the development cycle, not necessarily the complete program. Regardless of such proofs, testing is used invariably to obtain confidence in the correctness of the application.

Testing is not a perfect process in that a program might contain errors despite the success of a set of tests. However, it is a process with direct impact on our confidence in the correctness of the application under test. Our confidence in the correctness of an application increases when an application passes a set of thoroughly designed and executed tests.

Verification might appear to be a perfect process as it promises to verify that a program is free from errors. However, a close look at verification reveals that it has its own weaknesses. The person who verified a program might have made mistakes in the verification process; there might be an incorrect assumption on the input conditions; incorrect assumptions might be made regarding the components that interface with the program, and so on. Thus, neither verification nor testing are perfect techniques for proving the correctness of programs.

It is often stated that programs are mathematical objects and must be verified using mathematical techniques of theorem proving. While one could treat a program as a mathematical object, one must also realize the tremendous complexity of this object that exists within the program and also in the environment in which it operates. It is this complexity that has prevented formal verification of programs such as the 5ESS switch software from AT&T, the various versions of the Windows operating system, and other monstrously complex programs. Of course, we all know that these programs are defective, but the fact remains that they are usable and provide value to users.

1.9 Defect Management

Defect management is a practice used in several organizations. Several methods, e.g., Orthogonal Defect Classification, are available to categorize defects.

Defect management is an integral part of a development and test process in many software development organizations. It is a subprocess of the development process. It entails the following: defect prevention, discovery, recording and reporting, classification, resolution, and prediction.

Defect prevention is achieved through a variety of processes and tools. For example, good coding techniques, unit test plans, and code inspections are all important elements of any defect prevention process. Defect discovery is the identification of defects in response to failures observed during dynamic testing or found during static testing. Discovering a defect often involves debugging the code under test.

Defects found are classified and recorded in a database. Classification becomes important in dealing with the defects. For example, defects classified as "high severity" will likely be attended to first by the developers than those classified as "low severity." A variety of defect classification schemes exist. Orthogonal defect classification, popularly known as ODC, is one such scheme. Defect classification assists an organization in measuring statistics such as the types of defects, their frequency, and their location in the development phase and document. These statistics are then input to the organization's process improvement team that analyzes the data, identifies areas of improvement in the development process, and recommends appropriate actions to higher management.

Each defect, when recorded, is marked as "open" indicating that it needs to be resolved. One or more developers are assigned to resolve the defect. Resolution requires careful scrutiny of the defect, identifying a fix if needed, implementing the fix, testing the fix, and finally closing the defect indicating that it has been resolved. It is not necessary that every recorded defect be resolved prior to release. Only defects that are considered critical to the company's business goals, which include quality goals, are resolved, others are left unresolved until later.

Defect prediction is another important aspect of defect management. Organizations often do source code analysis to predict how many defects an application might contain before it enters the testing phase. Despite the imprecise nature of such early predictions, they are used to plan for testing resources and release dates. Advanced statistical techniques are used to predict defects during the test process. The predictions tend to be more accurate than early predictions due to the availability of defect data and the use of sophisticated models. The defect discovery data, including time of discovery and type of defect, are used to predict the count of remaining defects. Once again this information is often imprecise, though nevertheless used in planning.

Several tools exist for recording defects, and computing and reporting defect-related statistics. Bugzilla, open source, and FogBugz, commercially available, are three such tools. They provide several features for defect management including defect recording, classification, and tracking. Several tools that compute complexity metrics also predict defects using code complexity.

1.10 Test Generation Strategies

One of the key tasks in any software test activity is the generation of test cases. The program under test is executed against the test cases to

Strategies for test generation abound. Very broadly these are classified as either black box or white box.

determine whether or not it conforms to the requirements. The question *How to generate test cases?* is answered in significant detail in Part II: Test Generation of this book. Here, we provide a brief overview of the various test generation strategies.

Any form of test generation uses a source document. In the most informal of test methods, the source document resides in the mind of the tester who generates tests based on a knowledge of the requirements. In some organizations, tests are generated using a mix of formal and informal methods often directly from the requirements document serving as the source. In some test processes, requirements serve as a source for the development of formal models used for test generation.

Figure 1.12 summarizes several strategies for test generation. The top row in this figure captures techniques that are applied directly to the requirements. These may be informal techniques that assign values to input variables without the use of any rigorous or formal methods. These could also be techniques that identify input variables, capture the relationship amongst these variables, and use formal techniques for test generation such as random test generation and cause–effect graphing. Several such techniques are described in Chapter 4.

Most test generation strategies use requirements as a base, directly or indirectly, to generate tests. However, some random testing techniques focus on specific behavioral characteristics of a program, e.g., crash or hang, and do not necessarily derive test cases from requirements.

Figure 1.12 Requirements, models, and test generation algorithms.

Another set of strategies falls under the category *model based test generation*. These strategies require that a subset of the requirements be modeled using a formal notation. Such a model is also known as a specification of a subset of requirements. The tests are then generated with the specification serving as the source. Finite state machines, statecharts, Petri nets, and timed input–output automata are some of the well known and used formal notations for modeling various subsets of requirements. The notations fall under the category *graphical* notations though textual equivalents also exist. Several other notations, such as sequence and activity diagrams in UML, also exist and are used as models of subsets of requirements.

Languages based on predicate logic as well as algebraic languages are also used to express subsets of requirements in a formal manner. Each of these notational tools have their strengths and weaknesses. Usually, for any large application, one often uses more than notation to express all requirements and generate tests. Algorithms for test generation using finite state models are described in Chapter 5.

There also exist techniques to generate tests directly from the code. Such techniques, fall under *code-based test generation*. These techniques are useful when enhancing existing tests based on test adequacy criteria. For example, suppose that program P has been tested against tests generated from a statechart specification. After the successful execution of all tests one finds that some of the branches in P have not been covered, that is, there are some conditions that have never evaluated to both true and false. One could now use code-based test generation techniques to generate tests, or modify existing ones, to generate new tests that force a condition to evaluate to true or false, assuming that the evaluations are feasible.

As the name implies, code-based test generation uses program code as an input to a test generation tool.

Code-based test generation techniques are also used during regression testing when there is often a need to reduce the size of the test suite, or prioritize tests, against which a regression test is to be performed. Such techniques take four inputs, the program to be regression tested P', the program P from which P' has been derived by making changes, an existing test suite T for P, and some run time information obtained by executing P against T. This run time information may include items such as statement and branch coverage. The test generation algorithm then uses this information to select tests from T that must be executed to test those parts of P' that have changed or are affected by the changes made to P. The resulting test suite is usually a subset of T. Techniques for the reduction in the size of a test suite for the purpose of regression testing are described in Chapter 9.

1.11 Static Testing

Static testing does not require code execution, dynamic testing does.

Static testing is carried out without executing the application under test. This is in contrast to dynamic testing that requires one or more executions of the application under test. Static testing is useful in that it may lead to the discovery of faults in the application, as well as ambiguities and errors in requirements and other application relation documents, at a relatively low cost. This is especially so when dynamic testing is expensive. Nevertheless, static testing is complementary to dynamic testing. Organizations often sacrifice static testing in favor of dynamic testing though this is not considered a good practice.

Static testing is best carried out by an individual who did not write the code, or by a team of individuals. A sample process of static testing is illustrated in Figure 1.13. The test team responsible for static testing has access to requirements documents, application, and all associated documents such as design document and user manuals. The team also has access to one or more static testing tools. A static testing tool takes the application code as input and generates a variety of data useful in the test process.

1.11.1 Walkthroughs

Code walkthrough, also known as peer code review, is used to review code and may be considered as a static testing technique.

Walkthroughs and inspections are an integral part of static testing. Walkthrough is an informal process to review any application-related document. For example, requirements are reviewed using a process termed requirements walkthrough. Code is reviewed using code walkthrough, also known as peer code review.

A walkthrough begins with a review plan agreed upon by all members of the team. Each item of the document, for example, a source code module, is reviewed with clearly stated objectives in view. A detailed report is generated that lists items of concern regarding the document reviewed.

Figure 1.13 Elements of static testing.

In requirements walkthrough, the test team must review the requirements document to ensure that the requirements match user needs, and are free from ambiguities and inconsistencies. Review of requirements also improves the understanding of the test team regarding what is desired of the application. Both functional and nonfunctional requirements are reviewed. A detailed report is generated that lists items of concern regarding the requirements.

1.11.2 Inspections

Inspection is a more formally defined process than a walkthrough. This term is usually associated with code. Several organizations consider formal code inspections as a tool to improve code quality at a lower cost than incurred when dynamic testing is used. Organizations have reported significant increases in productivity and software quality due to the use of code inspections.

Code inspection is a rigorous process for assessing the quality of code.

Code inspection is carried out by a team. The team works according to an inspection plan that consists of the following elements: (a) statement of purpose, (b) work product to be inspected, this includes code and associated documents needed for inspection, (c) team formation, roles, and tasks to be performed, (d) rate at which the inspection task is to be completed, and (e) data collection forms where the team will record its findings such as defects discovered, coding standard violations, and time spent in each task.

Members of the inspection team are assigned roles of moderator, reader, recorder, and author. The moderator is in charge of the process and leads the review. Actual code is read by the reader, perhaps with the help of a code browser and with monitors for all in the team to view the code. The recorder records any errors discovered or issues to be looked into. The author is the actual developer whose primary task is to help others to understand the code. It is important that the inspection process be friendly and nonconfrontational. Several books and articles, cited in the Bibliography section, describe various elements of the code inspection process in detail.

1.11.3 Software complexity and static testing

Often a team must decide which of the several modules should be inspected first. Several parameters enter this decision-making process—one of these being module complexity. A more complex module is likely to have more errors and must be accorded higher priority for inspection than a module with lower complexity.

Static analysis tools often compute complexity metrics using one or more complexity metrics discussed in Section 1.6. Such metrics could be

used as a parameter in deciding which modules to inspect first. Certainly, the criticality of the function a module serves in an application could override the complexity metric while prioritizing modules.

1.12 Model-Based Testing and Model Checking

Model-based testing refers to the acts of modeling and the generation of tests from a formal model of application behavior. Model checking refers to a class of techniques that allow the validation of one or more properties from a given model of an application.

Model-based testing refers to the generation of tests from a model of the program generally derived from the requirements. Model checking is a formal verification process that checks whether or not a specific desired property of a program is satisfied.

Figure 1.14 illustrates the process of model checking. A model, usually in finite state, is extracted from some source. The source could be the requirements and, in some cases, the application code itself. Each state of the finite state model is prefixed with one or more properties that must hold when the application is in that state. For example, a property could be as simple as "$x < 0$" indicating that variable x must hold a negative value in this state. More complex properties, such as those related to timing, may also be associated.

One or more desired properties are then coded in a formal specification language. Often, such properties are coded in temporal logic, a language for formally specifying timing properties. The model and the desired properties are then input to a model checker. The model checker attempts to verify whether or not the given properties are satisfied by the given model.

For each property, the checker could come up with one of three possible answers: the property is satisfied, the property is not satisfied, or unable to determine. In the second case, the model checker provides a counter-example showing why the property is not satisfied. The third case might arise when the model checker is unable to terminate after an upper limit on the number of iterations has reached.

Figure 1.14 Elements of model checking.

In almost all cases, a model is a simplified version of the actual system requirements. A positive verdict by the model checker does not necessarily imply that the stated property is indeed satisfied in all cases. Hence, the need for testing. Despite a positive verdict by the model checker, testing is necessary to ascertain at least for a given set of situations that the application indeed satisfies the property.

While both model checking and model-based testing use models, model checking uses finite state models augmented with local properties that must hold at individual states. The local properties are known as atomic propositions and the augmented models as Kripke structures.

In summary, model checking is to be viewed as a powerful and complementary technique to model-based testing. Neither can guarantee whether an application satisfies a property under all input conditions. However, both point to useful information that helps a tester to discover sometimes subtle errors.

1.13 Types of Testing

An answer to the question "What types of testing are performed in your organization?" often consists of a set of terms such as black-box testing, reliability testing, unit testing, and so on. There exist a number of terms that typify one or more types of testing. We abstract all these terms as "X-testing." In this section, we present a framework for the classification of testing techniques. We then use this framework to classify a variety of testing techniques by giving meaning to the "X" in "X-testing."

Our framework consists of a set of four classifiers that serve to classify testing techniques that fall under the "dynamic testing" category. Techniques that fall under the "static testing"category are discussed in Section 1.11. Dynamic testing requires the execution of the program under test. Static testing consists of techniques for the review and analysis of the program.

Each of the four classifiers is a mapping from a set of features to a set of testing techniques. Features include source of test generation, questions that define a goal, a phase of life cycle or an artifact. Here are the four classifiers labeled as C1 through C4.

1. C1: Source of test generation
2. C2: Life cycle phase in which testing takes place
3. C3: Goal of a specific testing activity
4. C4: Characteristics of the artifact under test
5. C5: Test process

Tables 1.4 through 1.8 list each classifier by specifying the mapping and provide a few examples where appropriate. While each classifier defines a mapping, there exists a hierarchy across mappings. For example, black-box testing is used in almost all goal-directed testing. As is evident from the table, each mapping is not necessarily one-to-one. For example, pair-wise testing could be used to design tests for an entire system or for a component.

Test techniques that fall under mapping C1 are more generic than others. Each technique under C1 is potentially applicable to meet the

Table 1.4 Classification of techniques for testing computer software. Classifier: C1: Source of test generation.

Artifact	Technique	Example
Requirements (informal)	Black-box	Ad-hoc testing Boundary value analysis Category partition Classification trees Cause–effect graphs Equivalence partitioning Partition testing Predicate testing Random testing Syntax testing
Code	White-box	Adequacy assessment Coverage testing Data-flow testing Domain testing Mutation testing Path testing Structural testing Test minimization
Requirements and code	Black-box and White-box	
Formal model: Graphical or mathematical specification	Model-based Specification	Statechart testing FSM testing Pairwise testing Syntax testing
Component interface	Interface testing	Interface mutation Pairwise testing

goal specified in C3 as well as to test the software objects in C4. For example, one could use pairwise testing as a means to generate tests in any goal-directed testing that falls within C3. Let us now examine each classifier in more detail.

1.13.1 Classifier: C1: Source of test generation

Black-box testing: Test generation is an essential part of testing; it is as wedded to testing as the Earth is to the Sun. There are a variety of ways to generate tests; some are listed in Table 1.4. Tests could be generated from informally or formally specified requirements and without the aid of the code that is under test. Such form of testing is commonly referred to as black-box testing. When the requirements are informally specified, one could use ad-hoc techniques or heuristics such as equivalence partitioning and boundary value analysis.

In "pure" black-box testing tests are generated from the requirements; code is executed but not used to create or enhance tests.

Model-based or specification-based testing: Model-based or specification-based testing occurs when the requirements are formally specified, as for example, using one or more mathematical or graphical notations such as Z, state charts, and an event sequence graph, and tests are generated using the formal specification. This is also a form of black box testing. As listed in Table 1.4, there are a variety of techniques for generating tests for black-box and model-based testing. Part II of this book introduces several of these.

White-box testing: White-box testing refers to the test activity wherein code is used in the generation of, or the assessment of, test cases. It is rare, and almost impossible, to use white-box testing in isolation. As a test case consists of both inputs and expected outputs, one must use requirements to generate test cases; the code is used as an additional artifact in the generation process. However, there are techniques for generating tests exclusively from code and the corresponding expected output from requirements. For example, tools are available to generate tests to distinguish all mutants of a program under test or generate tests that force the program under test to exercise a given path. In any case, when someone claims they are using white-box testing, it is reasonable to conclude that they are using some forms of both black-box and white-box testing.

In white-box testing, code is generally used to enhance the quality of tests; the code itself serves to ascertain the "goodness" of tests.

Code could be used directly or indirectly for test generation. In the direct case, a tool, or a human tester, examines the code and focuses on a given path to be covered. A test is generated to cover this path. In the indirect case, tests generated using some black-box technique are assessed against some code-based coverage criterion. Additional tests are then generated to cover the uncovered portions of the code by analyzing which parts of the

code are feasible. Control flow, data flow, and mutating testing can be used for direct as well as indirect code-based test generation.

Interface testing is a form of black-box testing where the focus is on testing a program via its interfaces.

Interface testing: Tests are often generated using a component's interface. Certainly, the interface itself forms a part of the component's requirements and hence this form of testing is black-box testing. However, the focus on interface leads us to consider interface testing in its own right. Techniques such as pairwise testing and interface mutation are used to generate tests from a component's interface specification. In pairwise testing, the set of values for each input is obtained from the component's requirement. In interface mutation, the interface itself, such as a function coded in C or a CORBA component written in an IDL, serves to extract the information needed to perform interface mutation. While pairwise testing is clearly a black-box testing technique, interface mutation is a white-box technique though it focuses on the interface-related elements of the component under test.

Ad-hoc testing is not to be confused with random testing. In ad-hoc testing, a tester generates tests from requirements but without the use of any systematic method. Random testing uses a systematic method to generate tests. Generation of tests using random testing requires modeling the input space and then sampling data from the input space randomly.

In summary, black-box and white-box are the two most fundamental test techniques; they form the foundation of software testing. Test generation and assessment techniques are at the foundation of software testing. All the remaining test techniques classified by C2 through C4 fall into either the black-box or the white-box category.

1.13.2 Classifier: C2: Life cycle phase

Testing activities take place throughout the software life cycle. Each artifact produced is often subject to testing at different levels of rigor and using different testing techniques. Testing is often categorized based on the phase in which it occurs. Table 1.5 lists various types of testing depending on the phase in which the activity occurs.

Programmers write code during the early coding phase. They test their code before it is integrated with other system components. This type of testing is referred to as unit testing. When units are integrated and a large component or a subsystem formed, one does integration testing of the subsystem. Eventually, when the entire system has been built, its testing is referred to as system testing.

Test phases mentioned above differ in their timing and focus. In unit testing, a programmer focuses on the unit or a small component that has been developed. The goal is to ensure that the unit functions correctly in

Table 1.5 Classification of techniques for testing computer software. Classifier: C2: Life cycle phase.

Phase	Technique
Coding	Unit testing
Integration	Integration testing
System integration	System testing
Maintenance	Regression testing
Post system, pre-release	Beta-testing

Unit testing is an early phase of software testing and refers to the testing of specific units of a larger software system. Unit testing could be white-box, or black-box, or both.

isolation. In integration testing, the goal is to ensure that a collection of components function as desired. Integration errors are often discovered at this stage. The goal of system testing is to ensure that all the desired functionality is in the system and works as per its requirements. Note that tests designed during unit testing are not likely to be used during integration and system testing. Similarly, tests designed for integration testing might not be useful for system testing.

Often a carefully selected set of customers are asked to test a system before release. This form of testing is referred to as beta-testing. In the case of contract software, the customer who contracted the development performs acceptability testing prior to making the final decision as to whether or not to purchase the application for deployment.

Errors reported by users of an application often lead to additional testing and debugging. Often times, changes made to an application are much smaller in their size when compared to the entire application thus obviating the need for a complete system test. In such situations, one performs a regression test. The goal of regression testing is to ensure that the modified system functions per its specifications. However, regression testing may be performed using a subset of the entire set of test cases used for system testing. Test cases selected for regression testing include those designed to test the modified code and any other code that might be affected by the modifications.

It is important to note that all black-box and white-box testing techniques mentioned in Table 1.4 are applicable during each life cycle phase when code is being tested. For example, one could use the pairwise testing technique to generate tests for integration testing. One could also use any white box technique, such as test assessment and enhancement, during regression testing.

1.13.3 Classifier: C3: Goal-directed testing

Goal-directed testing leads to a large number of terms in software testing. Table 1.6 lists a sample of goals commonly used in practice, and the names of the corresponding test techniques.

There exists a variety of goals. Of course, finding any hidden errors is the prime goal of testing; goal-oriented testing looks for specific types of

Table 1.6 Classification of techniques for testing computer software. Classifier: C3: Goal-directed testing.

Goal	Testing Technique	Example
Advertised features	Functional	
Security	Security	
Invalid inputs	Robustness	
Vulnerabilities	Vulnerability	Penetration testing
Errors in GUI	GUI	Capture/playback Event sequence graphs Complete Interaction Sequence
Operational correctness	Operational	Transactional-flow
Reliability assessment	Reliability	
Resistance to penetration	Penetration	
System performance	Performance	Stress testing
Customer acceptability	Acceptance	
Business compatibility	Compatibility	Interface testing Installation testing
Peripherals compatibility	Configuration	
Foreign language compatibility	Foreign language	

Each goal is listed briefly. It is easy to examine each listing by adding an appropriate prefix such as "Check for," "Perform," "Evaluate," and "Check against." For example, the goal "Vulnerabilities" is to be read as "Check for Vulnerabilities."

failures. For example, the goal of vulnerability testing is to detect if there is any way by which the system under test can be penetrated by unauthorized users. Suppose that an organization has set up security policies and taken security measures. Penetration testing aims at evaluating how good these policies and measures are. Again, both black-box and white box techniques for the generation and evaluation of tests are applicable to penetration testing. Nevertheless, in many organizations, penetration and other forms of security testing remain ad-hoc.

Robustness testing aims to assess the robustness of an application against unintended, or invalid, inputs.

Robustness testing: Robustness testing refers to the task of testing an application for robustness against unintended inputs. It differs from functional testing in that the tests for robustness are derived from outside of the valid (or expected) input space whereas in the former the tests are derived from the valid input space.

As an example of robustness testing, suppose that an application is required to perform a function for all values of $x \geq 0$. However, there is no specification of what the application must do for $x < 0$. In this case, robustness testing would require that the application be tested against tests where $x < 0$. As the requirements do not specify behavior outside the valid input space, a clear understanding of what robustness means is needed for each application. In some applications, robustness might simply mean that the application displays an error message and exits, while in others, it might mean that the application brings an aircraft to a safe landing!

Stress testing tests an application for behavior under stress such as a large number of inputs or requests to a web server.

Stress testing: In stress testing, one checks for the behavior of an application under stress. Handling of overflow of data storage, for example, buffers, can be checked with the help of stress testing. Web applications can be tested by "stressing" them with a large number and variety of requests. The goal here is to find if the application continues to function correctly under stress.

One needs to quantify "stress" in the context of each application. For example, a web service can be stressed by sending it an unusually large number of requests. The goal of such testing would be to check if the application continues to function correctly and performs its services at the desired rate. Stress testing checks an application for conformance to its functional requirements as well as to its performance requirements when under stress.

Performance testing is used to test whether or not an application meets its performance requirements, e.g., the processing of online requests in a given amount of time.

Performance testing: The term "performance testing" refers to that phase of testing where an application is tested specifically with performance requirements in view. For example, a compiler might be tested to check if it meets the performance requirements stated in terms of the number of lines of code compiled per second.

Often performance requirements are stated with respect to a hardware and software configuration. For example, an application might be required

to process 1000 billing transactions per minute on a specific Intel processor based machine and running a specific operating system.

Load testing: The term "load testing" refers to that phase of testing in which an application is "loaded" with respect to one or more operations. The goal is to determine if the application continues to perform as required under various load conditions. For example, a database server can be loaded with requests from a large number of simulated users. While the server might work correctly when one or two users use it, it might fail in various ways when the number of users exceeds a threshold.

During load testing, one can determine whether or not the application is handling exceptions in an adequate manner. For example, an application might maintain a dynamically allocated buffer to store user IDs. The buffer size increases with an increase in the number of simultaneous users. There might be a situation when additional memory space is not available to add to the buffer. A poorly designed, or incorrectly coded, application might crash in such a situation. However, a carefully designed and correctly coded application will handle the "out of memory" exception in a graceful manner such as by announcing the high load through an apology message.

In a sense, load testing is yet another term for stress testing. However, load testing can be performed for ascertaining an application's performance as well as the correctness of its behavior. It is a form of stress testing when the objective is to check for correctness with respect to the functional requirements under load. It is a form of performance testing when the objective is to assess the time it takes to perform certain operations under a range of given conditions.

Load testing is also a form of robustness testing, i.e., testing for unspecified requirements. For example, there might not be any explicitly stated requirement related to the maximum number of users an application can handle and what the application must do when the number of users exceeds a threshold. In such situations, load testing allows a tester to determine the threshold beyond which the application under test fails through, e.g. a crash.

Terminology overlap: Note that there is some overlap in the terminology. For example, vulnerability testing is a form of security testing. Also, testing for compatibility with business goals might also include vulnerability testing. Such overlaps abound in testing-related terminology.

Once again note that formal techniques for test generation and test adequacy assessment apply to all forms of goal-directed testing. A lack of examples in almost the entire Table 1.6 is because test generation and assessment techniques listed in Table 1.4 are applicable to goal-directed

testing. It is technically correct to confess "We do ad-hoc testing" when none of the formal test generation techniques are used.

1.13.4 Classifier: C4: Artifact under test

Testers often say "We do X-testing" where X corresponds to an artifact under test. Table 1.7 is a partial list of testing techniques named after the artifact that is being tested. For example, during the design phase one might generate a design using the SDL notation. This design can be tested before it is committed to code. This form of testing is known as *design testing*.

As another example, you might have seen articles and books on OO-testing. Again, OO-testing refers to the testing of programs that are written in an object-oriented language such as C++ or Java. Yet again, there exists a variety of test generation and adequacy assessment

Table 1.7 Classification of techniques for testing computer software. Classifier: C4: Artifact under test.

Characteristic	Technique
Application component	Component testing
Batch processing	Equivalence partitioning, finite state model based testing, and most other test generation techniques discussed in this book
Client and server	Client-server testing
Compiler	Compiler testing
Design	Design testing
Code	Code testing
Database system	Transaction-flow testing
OO software	OO testing
Operating system	Operating system testing
Real-time software	Real-time testing
Requirements	Requirements testing
Web service	Web service testing

techniques that fall under black-box and white-box categories and that are applicable to the testing of OO software.

It is important to note that specialized black-box and white-box test techniques are available for specialized software. For example, timed-automata and Petri net-based test generation techniques are intended for the generation of tests for real-time software.

Batch processing applications pose a special challenge in testing. Payroll processing in organization and student record processing in academic institutions are two examples of batch processing. Often, these applications perform the same set of tasks on a large number records, for example, an employee record or a student record. Using the notions of equivalence partitioning and other requirements based test generation techniques discussed in Chapter 3, one must first ensure that each record is processed correctly. In addition, one must test for performance under heavy load, which can be done using load testing. While testing a batch processing application, it is also important to include an oracle that will check the result of executing each test script. This oracle might be a part of the test script itself. It could, for example, query the contents of a database after performing an operation that is intended to change the status of the database.

Sometimes an application might not be a batch processing application though a number of tests may need to be run in a batch. An embedded application, such as a cardiac pacemaker, is where there is a need to develop a set of tests that need to be run in a batch. Often, organizations develop a specialized tool to run a set of tests as a batch. For example, the tests might be encoded as scripts. The tool takes each test script and applies the test to the application. In a situation like this, the tool must have facilities such as to interrupt a test, suspend a test, resume a test, check test status, and schedule the batch test. IBM's WebSphere Studio is one of the several tools available for the development and testing of batch processing applications built using the J2EE environment.

1.13.5 Classifier: C5: Test process models

Software testing can be integrated into the software development life cycle in a variety of ways. This leads to various models for the test process listed in Table 1.8. Some of the models are described next.

Testing in the waterfall model: The waterfall model is one of the earliest, and least used, software life cycle models. Figure 1.15 shows the different phases in a development process based on the waterfall model. While verification and validation of documents produced in each phase is an essential activity, static as well as dynamic testing occurs toward

Table 1.8 Classification of techniques for testing computer software. Classifier: C5: Test process models.

Process	Attributes
Testing in waterfall model	Usually done toward the end of the development cycle.
Testing in V model	Explicitly specifies testing activities in each phase of the development cycle.
Spiral testing	Applied to software increments, each increment might be a prototype that eventually leads to the application delivered to the customer. Proposed for evolutionary software development.
Agile testing	Used in agile development methodologies such as extreme programming (XP).
Test Driven Development (TDD)	Requirements specified as tests.

the end of the process. Further, as the waterfall model requires adherence to an inherently sequential process, defects introduced in the early phases and discovered in later phases could be costly to correct. There is very little iterative or incremental development when using the waterfall model.

Testing in the V model: The V-model, as shown in Figure 1.16, explicitly specifies testing activities associated with each phase of the development cycle. These activities begin from the start and continue until the end of the life cycle. The testing activities are carried out in parallel with the development activities. Note that the V-model consists of the same development phases as in the waterfall model; the visual layout and an explicit specification of the test activities are the key differentiators. It is also important to note that test design begins soon after the requirements are available.

Spiral testing: The term "spiral testing" is not to be confused with spiral model, though they are both similar as both can be visually represented as a spiral of activities as in Figure 1.17. The spiral model is a generic model that can be used to derive process models such as the waterfall model, the V-model, and the incremental development model. While testing is a key activity in the spiral model, spiral testing refers to a test strategy that can be applied to any incremental software development process especially where

Spiral testing is generally used in an incremental process development model and leads to testing in all phases of the application development.

Figure 1.15 Testing in the waterfall model. Arrows indicate the "flow" of documents from one to the next. For example, design documents are input to the coding phase. The waterfall nature of the flow led to the name of this model.

Figure 1.16 Testing in the V model.

Figure 1.17 A visual representation of spiral testing. Test activities evolve over time and with the prototype. In the final iteration, the application is available for system and acceptance testing.

a prototype evolves into an application. In spiral testing the sophistication of test activities increases with the stages of an evolving prototype.

In the early stages, when a prototype is used to evaluate how an application must evolve, one focuses on test planning. The focus at this stage is on how testing will be performed in the remainder of the project. Subsequent iterations refine the prototype based on a more precise set of requirements. Further test planning takes place and unit and integration tests are performed. In the final stage, when the requirements are well defined, testers focus on system and acceptance testing. Note that all test generation techniques described in this book are applicable in the spiral testing paradigm. Note from Figure 1.17 that the cost of the testing activities (vertical axis) increases with subsequent iterations.

Agile testing: This is a name given to a test process that is rarely well defined. One way to define it is to specify what agile testing involves in addition to the usual steps such as test planning, test design, and test execution. Agile testing promotes the following ideas: (a) include testing related activities throughout a development project starting from the requirements phase, (b) work collaboratively with the customer who specifies requirements in terms of tests, (c) testers and developers must collaborate with each other rather than serve as adversaries, and (d) test often and in small chunks.

While there exist a variety of models for the test process, the test generation and adequacy techniques described in this are applicable to all.

Certainly, focus on test process is an important aspect of the management of any software development process.

The next example illustrates how some of the different types of testing techniques can be applied to test the same piece of software. The techniques so used can be easily classified using one or more of the classifiers described above.

Example 1.21 Consider a web service W to be tested. When executed, W converts a given value of temperature from one scale to another, for example from the Fahrenheit scale to the Celsius scale. Regardless of the technique used for test generation, we can refer to the testing of W as web-services testing. This reference uses the C4 classifier to describe the type of testing.

Next, let us examine various types of test generation techniques that could be used for testing W. First, suppose that tester A tests W by supplying sample inputs and checking the outputs. No specific method is used to generate the inputs. Using classifier C1, one may say that A has performed black-box testing and used an ad-hoc, or exploratory, method for test data generation. Using classifier C2 one may say that A has performed unit testing on W, assuming that W is a unit of a larger application. Given that W has a GUI to interface with a user, one can use classifier C3 and say that A has performed GUI testing.

Now suppose that another tester B writes a set of formal specifications for W using the Z notation. The tester generates, and uses, tests from the specification using techniques described in Volume 2. In this case, again using classifier C1, one may say that tester B has performed black-box testing and used a set of specification-based algorithms for test data generation.

Let us assume that we have a smarter tester C who generates tests using the formal specifications for W. C then tests W and evaluates the code coverage using one of the several code coverage criteria. C finds that the code coverage is not 100%, that is, some parts of the code inside W have remained uncovered, that is, untested, by the tests generated using the formal specification. C then generates and runs additional tests with the objective of exercising the uncovered portions of W. We say that here, C has performed both black-box and white-box testing. C has used specification-based test generation and enhanced the tests so generated to satisfy some control-flow based code coverage criteria.

Now, suppose that tester D tests W as a component of a larger application. Tester D does not have access to the code for W and

hence uses only its interface, and interface mutation, to generate tests. Using classifier C1 one may say that tester D has performed black-box testing and used interface mutation to generate tests (also see Exercise 1.17).

It should be obvious from the above example that simply using one classifier to describe a test technique might not provide sufficient information regarding details of the testing performed. To describe the set of testing techniques used to test any software object, one must clearly describe the following:

1. Test generation methods used; number of tests generated; number of tests run; number of tests failed and number passed.
2. Test adequacy assessment criteria used; results of test assessment stated in quantitative terms.
3. Test enhancement: number of additional tests generated based on the outcome adequacy assessment; number of additional tests run; number of additional failures discovered.

Note that test generation, adequacy assessment, and enhancement must be treated as a set of integrated activities. It is the sophistication of these activities, and their execution, that constitutes one important determinant of the quality of the delivered product.

The saturation effect is an empirical observation that additional test effort becomes useless once a test process has reached the saturation region. An understanding of the saturation effect aids a test team in planning when to switch test strategies to incrementally improve the quality of an application.

1.14 The Saturation Effect

The saturation effect is an abstraction of a phenomenon observed during the testing of complex software systems. Refer to Figure 1.18 to understand this important effect. The horizontal axis in the figure indicates the test effort that increases over time. The test effort can be measured as, for example, the number of test cases executed or total person days spent during the test and debug phase. The vertical axis refers to the true reliability (solid lines) and the confidence in the correct behavior (dotted lines) of the application under test. Note that the application under test evolves with an increase in test effort due to error correction.

The vertical axis can also be labeled as the "cumulative count of failures" that are observed over time, that is, as the test effort increases. The error correction process usually removes the cause of one or more failures. However, as the test effort increases additional failures may be found that causes the cumulative failure count to increase though it saturates as shown in the figure.

Figure 1.18 The saturation effect observed during the testing of complex software systems.

1.14.1 Confidence and true reliability

"Confidence" in Figure 1.18 refers to the confidence of the test manager in the true reliability of the application under test. An estimate of reliability, obtained using a suitable statistical method, can be used as a measure of confidence. "Reliability" in the figure refers to the probability of failure-free operation of the application under test in its intended environment. The "true" reliability differs from the estimated reliability in that the latter is an estimate of the application reliability obtained by using one of the many statistical methods. A 0 indicates lowest possible confidence and 1 the highest possible confidence. Similarly, a 0 indicates the lowest possible true reliability and a 1 indicates the highest possible true reliability.

1.14.2 Saturation region

Now suppose that application A is in the system test phase. The test team needs to generate tests, encapsulate them into appropriate scripts, set up the test environment, and run A against the tests. Let us assume that the tests are generated using a suitable test generation method (referred to as TGAT1 in Figure 1.18) and that each test either passes or fails. Each failure is analyzed and fixed, perhaps by the development team, if it is determined that A must not be shipped to the customer without fixing the cause of this failure. Whether the failure is fixed soon or later after it is detected, does not concern us in this discussion.

The true reliability of A, with respect to the operational profile used in testing, increases as errors are removed. Certainly, the true reliability could decrease in cases where fixing an error introduces additional errors–a case that we ignore in this discussion. If we measure the test effort as the combined effort of testing, debugging, and fixing the errors, the true reliability increases as shown in Figure 1.18 and eventually saturates, that is, stops increasing. Thus, regardless of the number of tests generated, and given the set of tests generated using TGAT1, the true reliability stops increasing after a certain amount of test effort has been spent. This saturation in the true reliability is shown in Figure 1.18 as the shaded region labeled SR1. Inside SR1, the true reliability remains constant while the test effort increases.

No new faults are found and fixed in the saturation region. Thus, the saturation region is indicative of wasted test effort under the assumption that A contains faults not detected while the test phase is in the saturation region.

1.14.3 False sense of confidence

The discovery and fixing of previously undiscovered faults might increase our confidence in the reliability of A. It also increases the true reliability of A. However, in the saturation region when the expenditure of test effort does not reveal new faults, the true reliability of A remains unchanged though our confidence is likely to increase due to a lack of observed failures. While in some cases this increase in confidence might be justified, in others it is a false sense of confidence.

This false sense of confidence is due to the lack of discovery of new faults, which in turn is due to the inability of the tests generated using TGAT1 to exercise the application code in ways significantly different from what has already been exercised. Thus, in the saturation region, the robust states of the application are being exercised, perhaps repeatedly, whereas the faults lie in other states.

The symbol Δ_1 in Figure 1.18 is a measure of the deviation from the true reliability of A and a test manager's confidence in the correctness of A. While one might suggest that Δ_1 can be estimated given an estimate of the confidence and the true reliability, in practice it is difficult to arrive at such an estimate due to the fuzzy nature of the confidence term. Hence, we must seek a quantitative metric that replaces human confidence.

1.14.4 Reducing Δ

Empirical studies reveal that every single test generation method has its limitations in that the resulting test set is unlikely to detect all faults in an

application. The more complex an application, the more unlikely it is that tests generated using any given method will detect all faults. This is one of the prime reasons why testers use, or must use, multiple techniques for test generation.

Suppose that a black-box test generation method is used to generate tests. For example, one might express the expected behavior of A using a finite state model and generate tests from the model as described in Chapter 5. Let us denote this method as TGAT1 and the test set so generated as T1.

Now, suppose that after having completed the test process using T1, we check how much of the code in A has been exercised. There are several ways to perform this check, and suppose that we use a control flow-based criterion, such as the MC/DC criterion described in Chapter 7. It is likely that T1 is inadequate with respect to the MC/DC criterion. Hence, we enhance T1 to generate T2, which is adequate with respect to the MC/DC criterion and $T1 \subset T2$. We refer to this second method of test generation as TGAT2.

Enhancement of tests based on feedback received from adequacy assessment leads to tests that are likely to move A to at least a few states that were never covered when testing using T1. This raises the possibility of detecting faults that might lie in the newly covered states. In the event, a failure occurs as a consequence of executing A against one or more tests in T2, the confidence exhibits a dip while the true reliability increases, assuming that the fault that caused the failure is removed and none introduced. Nevertheless, the true reliability once again reaches the saturation region, this time SR2.

The process described above for test enhancement can now be repeated using a different and perhaps a more powerful test adequacy criterion. The new test generation and enhancement technique is referred to in Figure 1.18 as TGAT3. The test set generated due to enhancement is T3 and $T2 \subset T3$. Once again we observe a dip in confidence, an increase in the true reliability, and eventually entry into the saturation region–SR3 in this case. Note that it is not always necessary that enhancement of a test set using an adequacy criteria will lead to a larger test set. It is certainly possible that T1=T2 or that T2=T3. However, this is unlikely to happen in a large application.

Theoretically the test generation and enhancement procedure described above can proceed almost forever especially when the input domain is astronomical in size. In practice, however, the process must terminate so that the application can be released. Regardless of when the application is released, it is likely that $\Delta > 0$ implying that while the confidence may be close to the true reliability, it is unlikely to be equal.

1.14.5 Impact on test process

A knowledge and appreciation of the saturation effect will likely be of value to any test team while designing and implementing a test process. Given that any method for the construction of functional tests will likely lead to a test set that is inadequate with respect to the code-based coverage criteria, it is important that tests be assessed for their goodness.

Various goodness measures are discussed in Chapters 7 and 8. It is some code coverage criterion that leads to saturation shown in Figure 1.18. For example, execution of additional tests might not lead to the coverage of any uncovered conditions. Enhancing tests using one or more code-based coverage criteria will likely help in moving out of a saturation region.

As the assessment of the goodness of tests does not scale well with the size of the application, it is important that this be done incrementally and using the application architecture. This aspect of test assessment is also covered in Chapters 7 and 8.

1.15 Principles of Testing

A principle, according to a definition, is a law that is usually followed. Principles can exist in several forms. A principle could be as a cause. For example, there exists principle of causality, or as a scientific law, a moral law, etc. Principles of software testing can be derived from experience, mathematical facts, and empirical studies. This section enumerates a few principles of software testing. In 2009, Bertrand Meyers postulated seven principles of software testing. The principles listed below are different from those proposed by Meyers, though one may find some similarity.

Principle 1: Definition

Testing is the activity of assessing how well a program behaves in relation to its expected behavior.

Complex software, such as an OS or a database system, can rarely be proven correct. Testing is the means to gain confidence in the correctness of the software. The level of confidence so gained depends on the individual viewing the test results as well as the number of successes and failures.

Principle 2: Reliability and confidence

Testing may increase one's confidence in the correctness of a program though the confidence may not match with the program's reliability.

A program's reliability is a statistical measure of its probability of successful operation in a given environment. It is estimated on a random test derived from an operational profile. Unless the operational profile is a close estimation of the expected usage scenario, a rare practical occurrence, the estimated reliability would likely be higher than the "true" reliability. Thus, while a diminishing rate of failures during testing increases reliability estimates, it may not necessarily increase the "true" reliability.

Principle 3: Coverage

A test case that tests untested portions of a program enhances or diminishes one's confidence in the program's correctness depending on whether the test passes or fails.

Principle 4: Orthogonality

Statistical measurements of the reliability of an application and the code coverage obtained by the tests used for measuring the reliability are orthogonal.

When an untested portion of the program passes a test, it increases one's confidence in the program correctness while a failure decreases the confidence. This implies that a good test suite consists of tests that test different portions of the code further implying that each test improves code coverage in some way.

Principle 5: Test suite quality

Code coverage is a reliable metric for the quality of a test suite.
Indeed, obtaining 100% code coverage does not imply correctness. However, tests that pass and improve code coverage lead to increased confidence in program correctness. A test suite adequate with respect to a coverage metric is likely to be better in terms of its error-revealing capability than another adequate with respect to a less powerful coverage metric.

Principle 6: Requirements and tests

Tests derived manually from requirements alone are rarely complete.

Complex systems are likely to have complex requirements. A complex set of requirements is difficult to understand in its entirety by any single individual. Thus, while a team of testers might generate tests to cover each requirement at least once, generating tests to cover the complex interactions across the individual requirements becomes an almost impossible task. This observation makes it important for test organizations to create one or more objective measures of test completeness.

Principle 7: Random testing

Random testing may or may not outperform nonrandom testing.

Random testing is a useful strategy to assess the robustness of a program. There is no guarantee that randomly generated tests will work better than a much smaller suite of tests generated using formal methods and aimed at functional testing. However, in many cases, random testing can outperform nonrandom testing especially when the aim is to check for any undesirable extreme behavior of a program such as a crash or a hang.

Principle 8: Saturation effect

The saturation effect is real and can be used as an effective tool for the improvement of test generation strategies.

Regardless of the test generation technique used, the rate at which latent faults are discovered is likely to be zero after a certain number of tests have been executed. To improve the "true" reliability of the program, it is best to switch to a different strategy for test generation.

Principle 9: Automation

Test automation aids in reducing the cost of testing and making it more reliable, but not necessarily in all situations.

Running a large number of tests manually and checking the outcome is indeed costly and error prone. Thus, test automation ought to be considered seriously in any system test environment. However, the cost of automating the test process needs to be weighed carefully against the benefits. While there exist several tools that automate the process of test generation, they do not overcome the problem of automating the process of generating an oracle that determines whether an automatically generated test passes or not.

Principle 10: Metrics

Integrating metrics into the entire test process aids in process improvement.

An organization-wide metrics program ought to include metrics related to all aspects of the test process. Such metrics, collected with help from tools, is helpful in assessing the improvement of, or otherwise, the quality of the test process and the released products.

Principle 11: Roles

Every developer is also a tester.

A developer is expected to write correct code, that is code that meets the requirements. While a developer could use model checking and other formal methods for proving the correctness of the written code, these are no substitutes for thorough testing. Thus, every developer ought to assume the role of a tester to ensure quality units that are to be integrated into a system.

1.16 Tools

A sample of tools useful for various types of testing that are not covered in any significant detail in this book is enumerated below. These include tools for performance, load, and stress testing, and for test management.

Test tools listed here is only a sample. It is best for a practitioner to be continually on the lookout for new and/or enhanced test tools.

Tool: WinRunner (HP Functional Testing)
Link: http://en.wikipedia.org/wiki/HP_Application_Lifecycle_Management#HP_Functional_Testing
Description

A well-known load-testing tool named "WinRunner" is now integrated into HP Functional Test suite. This suite consists of several tools for business process testing, functional, load, performance testing as well as for test data management.

Tool: JMeter
Link: http://jmeter.apache.org/
Description

JMeter is an open source Java application intended for load testing. It can load a variety of servers including HTTP, HTTPS, SOAP, and IMAP. It is multithreaded and hence allows concurrent sampling of various testing functions. JMeter has a GUI through which a test is controlled. JMeter serves as a master controller that controls multiple slave JMeter servers. Each server can send test commands to the target systems under test. Thus, JMeter serves to perform distributed testing.

Tool: JavaMetrics
Link: http://www.semdesigns.com/products/metrics/JavaMetrics.html
Description

JavaMetrics is sold by Semantic Designs. It calculates a variety of metrics from Java programs. The metrics include Source Lines of Code (SLOC), number of methods, decision density, cyclomatic complexity, and others. The same company makes similar tool for other languages including C#, COBOL, and VBScript.

Tool: COCOMO II
Link: http://diana.nps.edu/madachy/tools/COCOMOII.php
Description

COCOMO (Constructive Cost Model) II is a tool for estimating software development effort and cost. This is an online tool that asks a user to

enter a set of parameters that describe the software under development. These parameters are divided into two categories: scale drivers and cost drivers. Scale drivers include parameters such as development flexibility and process maturity. Cost drivers include parameters such as product complexity, programmer capability, application experience, and multisite development.

`Tool:` Bugzilla
`Link: http://www.bugzilla.org/`
`Description`

Bugzilla is a freely available application for tracking defects. It allows developers and testers to keep track of defects in their application. It allows communications among team members. The Mozilla Foundation, developer of Firefox, uses Bugzilla regularly.

`Tool:` Testopia
`Link: http://www.mozilla.org/projects/testopia/`
`Description`

This tool is an extension of the bug reporting and management tool Bugzilla. It allows the integration of test results with bug reports.

`Tool:` IBM Rational ClearCase
`Link: http://en.wikipedia.org/wiki/IBM_Rational_ClearCase`
`Description`

ClearCase is a family of tools that support test configuration management. It can be used to specify configurations using one of two use models: Unified Change Management (UCM) and base ClearCase.

SUMMARY

In this chapter, we have presented some basic concepts and terminology likely to be encountered by any tester. The chapter begins with a brief introduction to errors and the reason for testing. Next, we define input domain, also known as input space, as an essential source of test cases needed to test any program. The notions of correctness, reliability, and operational profiles are dealt with next.

Testing requires the generation and specification of tests. Section 1.5 covers this topic in some detail. The interaction between testing and debugging is also explained in this section. The IEEE standard for the specification of tests is also covered in this section.

Model-based testing is explained in Section 1.10 and thoroughly covered in Part II of this book. A number of organizations resort to model-based testing for various reasons such as the desire for high quality and automation of the test generation process.

Section 1.13 presents a framework for classifying a large number of testing techniques. The classification also helps in understanding what lie at the "foundation of software testing". Our view is that test generation, assessment, and enhancement lie at the foundation of software testing. It is this foundation that forms the focus of this book. We hope that the material in this section will help you to understand the complex relationship between various testing techniques and their use in different test scenarios.

Section 1.14 introduces an important observation from large-scale test processes. This observation, named as the saturation effect, might lead to a false sense of confidence in application reliability. Hence, it is important to understand the saturation effect, its impact on software reliability, and ways to overcome its shortcomings. Section 1.15 lists principles of testing that the students and practitioners of software testing might find useful. Lastly, Section 1.16 lists some tools useful in the test process.

Exercises

1.1 The following statement is often encountered: "It is impossible to test a program completely." Discuss the context in which this statement is true. Under what definition of "completely" is the statement true? (Note: Try answering this question now and return to it after having read Chapter 7.)

1.2 Describe at least one case in which a "perceived" incorrect behavior of a product is not due to an error in the product.

1.3 How many years will it take to test `max` exhaustively on a computer that takes 1 picosecond ($=10^{-12}$ seconds) to input a pair of integers and execute *max*?

1.4 It is required to develop a strategy to test whether or not a joke is excellent, good, or poor. What strategy would you propose? Would statistical techniques be useful in categorizing a joke?

1.5 Estimate the reliability of P of Example 1.12 given that (a) the input pair $(0, 0)$ appears with probability 0.6 and the remaining two pairs appear with probability 0.2 each and (b) P fails on the input $(-1, 1)$.

1.6 According to the ANSI/IEEE Std 729-1983, is it possible for a program to have a reliability of 1.0 even when it is infected with a number of defects? Explain your answer.

1.7 Suppose that the `sort` program in Example 1.13 is known to fail with probability 0.9 when alphanumeric strings are input and that it functions correctly on all numeric inputs. Find the reliability of `sort` with respect to the two operational profiles in Example 1.13.

1.8 Suggest one possible coding of `sort` that will lead to the behavior given in Example 1.9.

1.9 Consider a Web site that allows its visitors to search for items in a database using arbitrary search strings. Suppose we focus our attention only on three operations allowed by this site: `search`, `previous`, and `next`. To invoke the search operation, a search string is typed and the `Go` button clicked. When the search results are displayed, the `Previous` and `Next` buttons show up and may be clicked to move to the next or the previous page of search results. Describe how an oracle will determine whether or not the three functions of the Web site are implemented correctly. Can this oracle be automated?

1.10 Suggest a procedure to estimate the efficiency of a test set.

1.11 Let T_1 and T_2 be test sets designed to test how well program P meets requirements R. Suppose that the efficiency of both test sets is the same. (a) What features of the two test sets would you use to decide which one to select?

1.12 (a) Compute the cyclomatic complexity of the CFG in Figure 2.2 of the next chapter. (b) Prove that for CFGs of structured programs, the cyclomatic complexity is the number of conditions plus one. A structured program is one that uses only single entry and single exit constructs. GOTO statements are not allowed in a structured program.

1.13 For the two programs P1.1 and P1.2 compute Halstead's software science metrics. Discuss the relative complexity of the two programs using the metrics you have computed. Note the following while computing the software science metrics: (i) each token in a program, excluding semicolon (;) and braces, is to be considered as either an operator or an operand, (ii) each declaration keyword is an operator and the variable declared an operand, Thus, for example, keywords such as `if` and `else`, and function calls, are operators and so are the traditional arithmetic and relational symbols such as $<$ and $+$.

1.14 Consider two applications A_1 and A_2. Suppose that the testability of these applications is measured using a static complexity measure such as the cyclomatic complexity. Now suppose that the cyclomatic complexity of both applications is the same. Construct an example to show that despite the same cyclomatic complexity the testability of A_1 and A_2 might differ significantly. (*Hint: Think of embedded applications.*)

1.15 Consider a 2-input NAND gate in Figure 1.11(a) with the following faults: (i) input A stuck at 0 and output O stuck at 1, (ii) input A stuck at 0 and output O stuck at 0, (iii) inputs A and B stuck at 1, (iv) inputs A and B stuck at 0, and (v) input A stuck at 1 and B stuck at 0. For each of the five cases enumerated above, does there exist a test vector that reveals that the gate is faulty?

1.16 Why would tests designed for testing a component of a system not be usable during system test?

1.17 Company X is renowned for making state-of-the-art steam boilers. The latest model of gas or oil-fired boiler comes with a boiler control package that is being marketed as an

ultra-reliable piece of software. The package provides software for precise control of various boiler parameters such as combustion. The package has a user-friendly GUI to help with setting the appropriate control parameters. Study the following description and identify the test techniques used.

When contacted, the company revealed that the requirements for portions of the entire package were formally specified using the statechart graphical notation. The remaining portions were specified using the Z notation. Several test teams were assembled to test each component of the package.

The GUI test team's responsibility was to ensure that the GUI functions correctly and is user-friendly. The statechart and the Z specification served as a source of tests for the entire system. The tests generated from these specifications were assessed for adequacy using the MC/DC control flow-based coverage criterion. New tests were added to ensure that all feasible conditions were covered and the MC/DC criterion was satisfied. In addition to the tests generated from formal and graphical specifications, one test team used combinatorial designs to generate tests that would exercise the software for a variety of combinations of input parameters.

Individual components of the package were tested separately prior to integration. For each component, tests were designed using the traditional techniques of equivalence partitioning and boundary value analysis. Cause–effect graphing was used for some components. System test was performed with the help of tests generated.

1.18 List a few reasons that underlie the existence of the saturation effect.

2

Preliminaries: Mathematical

CONTENTS

2.1 Predicates and Boolean expressions
2.2 Control flow graph
2.3 Execution history
2.4 Dominators and post-dominators
2.5 Program dependence graph
2.6 Strings, languages, and regular expressions
2.7 Tools

The purpose of this introductory chapter is to familiarize the reader with the basic mathematical concepts related to software testing. These concepts will likely be useful in understanding the material in subsequent chapters.

2.1 Predicates and Boolean Expressions

A predicate is a Boolean function, also known as a condition, that evaluates to true or false. It consists of relational operators such as "<" and ">", and Boolean connectors such as "and" and "or"

Let *relop* denote a relational operator in the set $\{<, >, \leq, \geq, =, \neq\}$. Let *bop* denote a Boolean operator in the set $\{\wedge, \vee, , \neg\}$ where $\wedge$, $\vee$, and are binary Boolean operators and $\neg$ is a unary Boolean operator. A *Boolean variable* takes on values from the set $\{true, false\}$. Given a Boolean variable a, $\neg a$ and $\overline{a}$ denote the complement of a.

A *relational expression* is an expression of the form e_1 *relop* e_2 where e_1 and e_2 are expressions that assume values from a finite or infinite set S. It should be possible to order the elements of S so that e_1 and e_2 can be compared using any one of the relational operators.

A predicate is a Boolean function that evaluates to true or false. It consists of relational symbols such as "<" and ">", and Boolean connectors such as "and" and "or". In computer programs such a function is also known as condition. A condition can be represented as a *simple predicate* or a *compound predicate*. A simple predicate is a Boolean variable or a relational expression where any one or more variables might be negated. A compound predicate is a simple predicate or an expression formed by joining two or more simple predicates, possibly negated, with a binary Boolean operator. Parentheses in any predicate indicate groupings of variables and relational expressions. Examples of predicates and other terms defined in this section are given in Table 2.1.

Table 2.1 Examples of terms defined in Section 2.1.

Item	Examples	Comment
Simple predicate	p $q \wedge r$ $a + b < c$	p, q, and r are Boolean variables; a, b, and c are integer variables.
Compound predicate	$\neg(a + b < c)$ $(a + b < c) \wedge (\neg p)$	Parentheses are used to make explicit the grouping of simple predicates.
Boolean expression	p, $\neg p$ $p \wedge q \vee r$	
Singular Boolean expression	$p \wedge q \vee \overline{r} \wedge s$	p, q, r, and s are Boolean variables.
Non-singular Boolean expression	$p \wedge q \vee \overline{r} \wedge p$	

A *Boolean expression* is composed of one or more Boolean variables joined by Boolean operators. In other words, a Boolean expression is a predicate that does not contain any relational expression. When it is clear from the context, we omit the $\wedge$ operator and use the $+$ sign instead of $\vee$. For example, Boolean expression $p \wedge q \vee \bar{r} \wedge s$ can be written as $pq + \bar{r}s$.

Note that the term pq is also known as a *product* of variables p and q. Similarly, the term rs is the product of Boolean variables r and s. The expression $pq + \bar{r}s$ is a sum of two product terms pq and $\bar{r}s$. We assume left-to-right associativity for operators. Also, `and` takes precedence over `or`.

Each occurrence of a Boolean variable, or its negation, inside a Boolean expression, is considered as literal. For example, p, q, r, and p are four literals in the expression $p \wedge q \vee \bar{r} \wedge p$.

A predicate p_r can be converted to a Boolean expression by replacing each relational expression in p_r by a distinct Boolean variable. For example, the predicate $(a + b < c) \wedge (\neg d)$ is equivalent to the Boolean expression $e_1 \wedge e_2$ where $e_1 = a + b < c$ and $e_2 = \neg d$.

A Boolean expression is considered *singular* if each variable in the expression occurs exactly once. Consider the following Boolean expression E that contains k distinct Boolean expressions $e_1, e_2, \ldots, e_k$.

A Boolean expression is considered singular if each variable in the expression occurs exactly once.

$$E = e_1 \; bop \; e_2 \; bop \; \ldots e_{k-1} \; bop \; e_k.$$

For $1 \leq i \leq k$, $1 \leq j \leq k$, $i \neq j$, we say that e_i and e_j are *mutually singular* if they do not share any variable. e_i is considered a singular component of E if and only if e_i is singular and is mutually singular with each of the other $k - 1$ components of E. e_i is considered non-singular if and only if it is non-singular by itself and is mutually singular with the remaining $k - 1$ elements of E.

A Boolean expression is considered to be in *disjunctive normal form* when it is expressed as a sum of product terms. It is in conjunctive normal form when expressed as a product of sums of terms. For example, $pq + \bar{r}s$ is a DNF expression. A Boolean expression is in *conjunctive normal form*, also referred to as CNF, if it is written as a product of sums. For example, the expression $(p + \bar{r})(p + s)(q + \bar{r})(q + s)$, is a CNF equivalent of expression $pq + \bar{r}s$. Note that any Boolean expression in CNF can be converted to an equivalent DNF and vice versa.

A Boolean expression is considered to be in disjunctive normal form when it is expressed as a sum of product terms. It is in conjunctive normal form when expressed as a product of sums of terms.

A Boolean expression can also be represented as an abstract syntax tree as shown in Figure 2.1. We use the term $AST(p_r)$ for the abstract syntax tree of a predicate p_r. Each leaf node of $AST(p_r)$ represents a Boolean variable or a relational expression. An internal node of $AST(p_r)$ is a Boolean operator such as $\wedge$, $\vee$, , and $\neg$, and is known as an *AND* node, *OR* node, *XOR* node, and a *NOT* node, respectively.

Figure 2.1 An abstract syntax tree for compound predicate $(a+b<c) \wedge (\neg p) \vee (r>s)$.

2.2 Control Flow Graph

A control flow graph (CFG) captures the flow of control in a program. It consists of basic blocks and directed arcs connecting the blocks.

A control flow graph (CFG) captures the flow of control in a program. It consists of basic bocks and directed arcs connecting the blocks. Such a graph assists testers in the analysis of a program to understand its behavior in terms of the flow of control. A control flow graph can be constructed manually without much difficulty for relatively small programs, say containing less than about 50 statements. However, as the size of the program grows, so does the difficulty of constructing its control flow graph and hence arises the need for tools.

A CFG is also known by the names *flow graph* or *program graph*. However, it is not to be confused with the program dependence graph introduced in Section 2.5. In this section, we explain what a control flow graph is and how to construct one for a given program.

2.2.1 Basic blocks

A basic block is the longest possible sequence of consecutive statements in a program such that control can enter the block only at the first statement and exit from the last. Thus, there cannot be any conditional statement inside a basic block other than at its end.

Let P denote a program written in a procedural programming language, be it high level as C or Java, or a low level such as the 80 × 86 assembly. A basic block is the longest possible sequence of consecutive statements in a program such that control can enter the block only at the first statement and exit from the last. Thus, there cannot be any conditional statement inside a basic block other than at its end. Thus, a block has unique entry and exit points. These points are the first and the last statements, respectively, within a basic block. Control always enters a basic block at its entry point and exits from its exit point. There is no possibility of exit or a halt at any point inside the basic block except at its exit point. The entry and exit points of a basic block coincide when the block contains only one statement.

Example 2.1 The following program takes two integers `x` and `y` and outputs x^y. There are a total of 17 lines in this program including the `begin` and `end`. The execution of this program begins at line 1 and moves through lines 2, 3, and 4 to line 5 containing an

`if` statement. Considering that there is a decision at line 5, control could go to one of two possible destinations at line 6 and line 8. Thus, the sequence of statements starting at line 1 and ending at line 5 constitutes a basic block. Its only entry point is at line 1 and the only exit point is at line 5.

Program P2.1

```
1   begin
2     int x, y, power;
3     float z;
4     input (x, y);
5     if (y<0)
6      power=-y;
7     else
8      power=y;
9     z=1;
10    while (power!=0){
11     z=z*x;
12     power=power-1;
13    }
14    if (y<0)
15     z=1/z;
16    output(z);
17  end
```

A list of all basic blocks in P2.1 is given below.

Block	Lines	Entry point	Exit point
1	2, 3, 4, 5	1	5
2	6	6	6
3	8	8	8
4	9	9	9
5	10	10	10
6	11, 12	11	12
7	14	14	14
8	15	15	15
9	16	16	16

P2.1 contains a total of 9 basic blocks numbered sequentially from 1 to 9.

Note how the `while` at line 10 forms a block of its own. Also note that we have ignored lines 7 and 13 from the listing because these are syntactic markers, and so are `begin` and `end` that are also ignored.

Note that some tools for program analyses place a procedure call statement in a separate basic block. If we were to do that then we will place the `input` and `output` statements in P2.1 in two separate basic blocks. Consider the following sequence of statements extracted from P2.1.

```
1   begin
2     int x, y, power;
3     float z;
4     input (x, y);
5     if (y<0)
```

In the previous example, lines 1 through 5 constitute one basic block. The above sequence contains a call to the `input` function. If function calls are treated differently, then the above sequence of statements contains three basic blocks, one comprising lines 1 through 3, the second comprising line 4, and the third comprising line 5.

Function calls are often treated as blocks of their own because they cause the control to be transferred away from the currently executing function and hence raise the possibility of abnormal termination of the program. In the context of flow graphs, unless stated otherwise, we treat calls to functions like any other sequential statement that is executed without the possibility of termination.

2.2.2 Flow graphs

A flow graph is another name for a CFG. It consists of nodes and directed edges. Each node represents a basic block in the program and each edge represents the flow of control from its source node to the destination node.

A flow graph G is defined as a finite set N of nodes and a finite set E of directed edges. A flow graph is another name for a CFG. It consists of nodes and directed edges. Each node represents a basic block in the program and each edge represents the flow of control from its source node to the destination node. An edge (i,j) in E, with an arrow directed from i to j, connects nodes n_i and n_j in N. We often write $G = (N, E)$ to denote a flow graph G with nodes in N and edges in E. `Start` and `End` are two special nodes in N and are known as distinguished nodes. Every other node in G is reachable from `Start`. Also, every node in N has a path

terminating at `End`. Node `Start` has no incoming edge and node `End` has no outgoing edge.

In a flow graph of program P, we often use a basic block as a node and edges indicate the flow of control across basic blocks. We label the blocks and the nodes such that block b_i corresponds to node n_i. An edge (i,j) connecting basic blocks b_i and b_j implies that control can go from block b_i to block b_j. Sometimes we will use a flow graph with one node corresponding to each statement in P.

A pictorial representation of a flow graph is often used in the analysis of the control behavior of a program. Each node is represented by a symbol, usually an oval or a rectangular box. These boxes are labeled by their corresponding block numbers. The boxes are connected by lines representing edges. Arrows are used to indicate the direction of flow. A block that ends in a decision has two edges going out of it. These edges are labeled `true` and `false` to indicate the path taken when the condition evaluates to `true` and `false`, respectively.

Example 2.2 The flow graph for P2.1 is defined as follows:

$$\begin{aligned} N &= \{\texttt{Start}, 1, 2, 3, 4, 5, 6, 7, 8, 9, \texttt{End}\} \\ E &= \{(\texttt{Start}, 1), (1,2), (1,3), (2,4), (3,4), (4,5), (5,6), \\ &\quad (6,5), (5,7), (7,8), (7,9), (8,9), (9, \texttt{End})\} \end{aligned}$$

Figure 2.2(a) depicts this flow graph. Block numbers are placed right next to or above the corresponding box. As shown in Figure 2.2(b), the contents of a block may be omitted, and nodes are represented by circles, if we are interested only in the flow of control across program blocks and not their contents.

2.2.3 Paths

Consider a flow graph $G = (N, E)$. A sequence of k edges, $k > 0$, $(e_1, e_2, \ldots, e_k)$, denotes a path of length k through the flow graph if the following sequence condition holds: Given that n_p, n_q, n_r, and n_s are nodes belonging to N, and $0 < i < k$, if $e_i = (n_p, n_q)$ and $e_{i+1} = (n_r, n_s)$ then $n_q = n_r$.

Thus, for example, the sequence ((1, 3), (3, 4), (4, 5)) is a path in the flow graph shown in Figure 2.2. A path through a flow graph is a sequence of edges. It indicates the flow of control in the corresponding program. Not all paths extracted from a flow graph might be feasible implying that there does not exist any test input that will exercise an infeasible path in the corresponding program. However, ((1, 3), (3, 5), (6, 8)) is not a valid path through this flow graph. For brevity, we indicate a path as a sequence of

A path through a flow graph is a sequence of edges. It indicates the flow of control in the corresponding program. Not all paths extracted from a flow graph might be feasible implying that there does not exist any test input that will exercise an infeasible path in the corresponding program.

Figure 2.2 Flow graph representations of Program P2.1. (a) Statements in each block are shown. (b) Statements within a block are omitted.

blocks. For example, in Figure 2.2, the edge sequence ((1, 3), (3, 4), (4, 5)) is the same as the block sequence (1, 3, 4, 5).

For nodes $n, m \in N$, m is said to be a *descendent* of n if there is a path from n to m; in this case, n is an *ancestor* of m and m its descendent. If, in addition, $n \neq m$, then n is a *proper ancestor* of m, and m a *proper descendent* of n. If there is an edge $(n, m) \in E$, then m is an *immediate successor* of n and n the *immediate predecessor* of m.

The set of all nodes that can be reached from node n along a path of length one or more is known as the *successor* set of node n. Similarly, the set of all nodes from which there exists a path of length one or more to node n is known as the *predecessor* set for node n. The set of successor and predecessor nodes of n is denoted as `succ`(n) and `pred`(n), respectively. The `Start` node has no ancestor and the `End` node has no descendent.

A path through a flow graph is considered *complete* if the first node along the path is `Start` and the terminating node is `End`. A path p through

a flow graph for program P is considered *feasible* if there exists at least one test case that when input to P causes p to be traversed. If no such test case exists, then p is considered *infeasible*. Whether or not a given path is feasible is an undecidable problem. This statement implies that it is not possible to write an algorithm that takes as inputs an arbitrary program and a path through that program, and correctly outputs whether or not this path is feasible for the given program.

A subsequence of a path is *subpath*. A subpath that starts with the start node is known as an *initial* subpath. Given paths $p = \{n_1, n_2, \ldots, n_t\}$ and $s = \{i_1, i_2, \ldots, i_u\}$, s is a subpath of p if for some $1 \leq j \leq t$ and $j + u - 1 \leq t$, $i_1 = n_j, i_2 = n_{j+1}, \ldots, i_u = n_{j+u-1}$. In this case, we also say that s and each node i_k for $1 \leq k \leq u$, are included in p. Edge (n_m, n_{m+1}) that connects nodes n_m and n_{m+1}, is considered included in p for $1 \leq m \leq (t-1)$.

Each path or subpath p in a CFG can be represented by a path vector of length $e = |E|$. A path vector is a list of n 0's and 1's. Each entry on the list corresponds to an edge in the flow graph. A 0 entry indicates that the corresponding edge does not lie along that path while a 1 indicates it does. Each entry in a path vector corresponds to an edge in the CFG and indicates the number of times that edge has been traversed along the path. Note that a path vector for a subpath will have 0's corresponding to the edges not included in that subpath. We shall denote a path vector as V_p.

A path vector is a list of n 0's and 1's. Each entry on the list corresponds to an edge in the flow graph. A 0 entry indicates that the corresponding edge does not lie along that path while a 1 indicates it does.

Example 2.3 In Figure 2.2, the following two are complete and feasible paths of lengths 10 and 9, respectively. The paths are listed using block numbers, the start node, and the terminating node. Figure 2.3 shows the first of these two complete paths:

$p_1 = (\texttt{Start}, 1, 2, 4, 5, 6, 5, 7, 8, 9, \texttt{End})$

$p_2 = (\texttt{Start}, 1, 3, 4, 5, 6, 5, 7, 9, \texttt{End})$

As there are 13 edges in Figure 2.2(b), the path vector consists of 13 entries, one corresponding to each edge in the set E in Example 2.2. Assuming that the first entry in the path vector corresponds to the first entry in E, the second to the second entry in E, and so on, we obtain the following path vectors for p_1 and p_2.

$V_{p_1} = [1101011111011]$

$V_{p_1} = [1010111110101]$

(4, 5, 6) is a subpath of p_1. (`Start`, 1, 2, 4, 5) is an initial subpath. The next two paths of lengths 3 and 4 are incomplete. The first of these two paths is shown by a dashed line in Figure 2.3.

$p_3 = (5, 7, 8, 9)$

$p_4 = (6, 5, 7, 9, \texttt{End})$

The next two paths of lengths 10 and 7 are complete but infeasible.

Figure 2.3 Flow graph representation of Program P2.1. A complete path is shown using bold edges and a sub-path using a dashed line.

$p_5 = (\texttt{Start}, 1, 3, 4, 5, 6, 5, 7, 8, 9, \texttt{End})$
$p_6 = (\texttt{Start}, 1, 2, 4, 5, 7, 9, \texttt{End})$
Finally, the next two paths are invalid as they do not satisfy the sequence condition stated earlier.
$p_7 = (\texttt{Start}, 1, 2, 4, 8, 9, \texttt{End})$
$p_8 = (\texttt{Start}, 1, 2, 4, 7, 9, \texttt{End})$
Nodes 2 and 3 in Figure 2.3 are successors of node 1, nodes 6 and 7 are successors of node 5, and nodes 8 and 9 are successors of node 7. Nodes 6, 7, 8, 9, and `End` are descendents of node 5. We also have `succ`(5)= {5, 6, 7, 8, 9, `End`} and `pred`(5)={`Start`, 1, 2 , 3, 4, 5, 6}. Note that in the presence of loops, a node can be its own ancestor or descendent.

Each additional condition in a program adds at least one path.

There can be many distinct paths through a program. Each additional condition in a program adds at least one path. A program with no condition contains exactly one path that begins at node `Start` and terminates at node `End`. However, each additional condition in the program increases

the number of distinct paths by at least one. Depending on their location, conditions can have an exponential effect on the number of paths.

Example 2.4 Consider a program that contains the following statement sequence with exactly one statement containing a condition. This program has two distinct paths: one that is traversed when C_1 is `true` and the other when the condition is `false`.

```
begin
  S1;
  S2;
  ⋮
  if (C1) {...}
  ⋮
  Sn;
end
```

We modify the program given above by adding another `if`. The modified program, shown below, has exactly four paths that correspond to four distinct combinations of conditions C_1 and C_2.

```
begin
  S1;
  S2;
  ⋮
  if (C1) {...}
  ⋮
  if (C2) {...}
  Sn;
end
```

Notice the exponential effect of adding an `if` on the number of paths. However, if a new condition is added within the scope of an `if` statement then the number of distinct paths increases only by 1 as is the case in the following program that has only three distinct paths.

```
begin
  S1;
  S2;
  ⋮
```

```
    if (C1) {
     ⋮
     if (C2) {...}
     ⋮
    }
    ⋮
    Sn;
  end
```

Loops can significantly increase the number of paths in a program making exhaustive testing nearly impossible. Some assumptions on the distribution of the inputs can be used to estimate the number of paths in a program in the presence of loops and thus derive a few tests to test loops.

The presence of loops can enormously increase the number of paths. Loops can significantly increase the number of paths in a program making exhaustive testing nearly impossible. Some assumptions on the distribution of the inputs can be used to estimate the number of paths in a program in the presence of loops and thus derive a few tests to test loops. Each traversal of the loop body adds a condition to the program thereby increasing the number of paths by at least one. Sometimes the number of times a loop is to be executed depends on the input data and cannot be determined prior to program execution. This becomes another cause of difficulty in determining the number of paths in a program. Of course, one can compute an upper limit on the number of paths based on some assumption on the input data.

Example 2.5 Program P2.2 inputs a sequence of integers and computes their product. A Boolean variable `done` controls the number of integers to be multiplied. A flow graph for this program appears in Figure 2.4.

Program P2.2

```
begin
int num, product;
bool done;
product = 1;
input(done);
while (!done){
   input(num);
   product = product*num;
   input(done);
}
output(product);
end
```

Figure 2.4 Flow graph of Program P2.2. Numbers 1 through 4 indicate the four basic blocks in P2.2.

As shown in Figure 2.4, Program P2.2 contains four basic blocks and one condition that guards the body of `while`. (`Start`, 1, 2, 4, `End`) is the path traversed when `done` is `true` the first time the loop condition is checked. If there is only one value of `num` to be processed, then the path followed is (`Start`, 1, 2, 3, 2, 4, `End`). When there are two input integers to be multiplied then the path traversed is (`Start`, 1, 2, 3, 2, 3, 2, 4, `End`).

Notice that the length of the path traversed increases with the number of times the loop body is traversed. Also, the number of distinct paths in this program is the same as the number of different lengths of input sequences that are to be multiplied. Thus, when the input sequence is empty, i.e., is of length 0, the length of the path traversed is 4. For an input sequence of length 1, it is 6, for 2 it is 8, and so on.

A set of linearly independent paths through a flow graph is known as a *basis path set*. Any path through the flow graph is a linear combination of the basis paths. The number of basis paths is equal to the cyclomatic complexity of the flow graph.

2.2.4 Basis paths

Let P denote a program and G its control flow graph. A basis path set consists of one or more linearly independent complete paths through G. Each path in this set is known as a *basis path*. A basis path set is also known as a *basis set*.

Let S denote the set of all paths in a CFG G and S_b the basis set consisting of n basis paths. Let p be any path in S, but not in S_b, and V_p its path vector. Let $V_1, V_2, \ldots V_n$ denote the path vectors of the basis paths. Then V_p can be represented as the following linear combination.

$V_p = a_1 V_1 + a_2 V_2 \ldots + a_n V_n$, where $a_1, a_2, \ldots a_n$ are integers at least one of which is non-zero.

A basis set is constructed from G. The first path in the set could be any complete path through G that starts at node `Start`, ends at node `End`, and does not iterate any loop more than once. The subsequent paths can be derived by changing the outcome of one of the conditions in any of the paths derived so far such that the new path is not identical to any already derived. A basis path either skips the loop body or iterates it exactly once.

A path vector for each basis path consists of only 0s and 1s as each edge along such a path is either not traversed or traversed exactly once. The path vectors for the flow graphs in Figure 2.5 are listed just below the paths in the Table 2.2.

The number of basis paths is equal to the cyclomatic complexity of G. Recall from Chapter 1 that the cyclomatic complexity $V(G)$ of a program with control graph G can be calculated by the following formula.

$$V(G) = e - n + 2p$$

where e is the number of edges, n the number of nodes, and p the number of connected components in G.

Figure 2.5 Sample control flow graphs to illustrate basis paths.

Table 2.2 Basis set and path vectors for the CFGs in Figure 2.5

CFG (G) (Figure 2.5)	Edges	Nodes	V(G)	Basis set path vectors
(a)	4	5	1	(`Start`, 1, 2, 3, `End`) [1 1 1 1]
(b)	6	6	2	(`Start`, 1, 2, 4, `End`) [1 1 0 1 0 1] (`Start`, 1, 3, 4, `End`) [1 0 1 0 1 1]
(c)	8	7	3	(`Start`, 1, `End`) [1 0 0 0 0 0 0 1] (`Start`, 1, 2, 3, 5, 1 `End`) [1 1 1 1 1 0 0 1] (`Start`, 1, 2, 4, 5, 1, `End`) [1 1 0 0 10 1 1 1]

Example 2.6 For each flow graph in Figure 2.5 the cyclomatic complexity, basis paths, and the corresponding path vectors are given in Table 2.2. The set of edges, and the corresponding sequencing of entries in the path vectors in this table are as follows.

Figure 2.5	Set of edges
(a)	{(`Start`, 1), (1, 2), (2, 3), (3, `End`)}
(b)	{`Start`, 1), (1, 2), (2, 4), (3, 4). (4,`End`)}
(c)	{`Start`, 1), (1, 2), (2, 3), (3, 5). (2, 4), (4, 5), (5, 1), (5,`End`)}

Exercises 2.3 through 2.6 ask you to derive basis sets for a few other programs.

2.2.5 Path conditions and domains

A *path condition* is a predicate that must evaluate to true for a path to be executed. For a straight line program having no conditional statements, the path condition is trivially *true*. However, for most programs, the path condition is a compound predicate obtained by conjoining the predicates along the path under consideration. A path condition partitions the input domain of a program. Thus, all inputs that satisfy the path condition are said to be in the *path domain*. The next example illustrates how to determine path conditions and hence the corresponding path domains.

Every path has an associated condition that is used to decide whether or not that path is executed. Such a condition is known as a *path condition*. A path is traversed only when the associated path condition evaluates to true.

All program inputs that satisfy a path condition are considered belonging to the domain of that path. A path domain is a subset of a program's input domain.

Example 2.7 Consider the following program that takes two non-negative integers $x1$ and $y1$ as inputs and computes their greatest common divisor.

```
1     int gcd(int x1, int y1){
2          int x, y;
3          input(x1,y1);
4          x=x1; y=y1;
5          while(x!=y){
6               if(x>y)
7                    x=x-y;
8               else
9                    y=y-x;
10         }
11         return x;
12     }
```

The above program has an infinite number of paths one each for a set of pairs of integers. This set constitutes the domain of that path. Following is a list of a few paths through this program. These paths are labeled p_0, p_{11}, and p_{12} where the first of the two indices denotes the number of times the loop is iterated and the second one indicates the variation within the loop due to the condition in the `if` statement.

Loop iterated zero times: p_0= (1, 2, 3, 4, 5, 11)
Loop iterated once:
For $x > y$: p_{11}: (1, 2, 3, 4, 5, 6, 7, 10, 5, 11)
For $x \leq y$: p_{12}: (1, 2, 3, 4, 5, 6, 9, 10, 5, 11)

A path domain can be computed by first finding the path conditions. The set of inputs that satisfy these conditions is the path domain for the path under consideration. For a program with no conditions the path domain is the input domain.

Note that each loop iteration leads to one additional path. This path depends on the code in the loop body. In the above example, the path could correspond to either the true or the false branch of the `if` statement.

To determine the path domains, let us compute the condition associate with each path. Following are the conditions, and their simplified versions, for each of the three paths above.

$p_0 : x1 = y1$
$p_{11} : (x1 \neq y1 \wedge (x1 - y1) = y1) \rightarrow x1 = 2 * y1$
$p_{12} : (x1 \neq y1 \wedge x1 = (y1 - x1)) \rightarrow 2 * x1 = y1$

Note that each path condition is expressed in terms of the program inputs $x1$ and $y1$. Thus, for example, the first time the loop

condition $x \neq y$ is evaluated, the values of program variables x and y are, respectively, $x1$ and $y1$. Hence the loop condition expressed in terms of input variables is $x1 \neq y1$ and the corresponding path condition is $x1 = y1$. Similarly, the interpreted path conditions for p_{11} and p_{12} are, respectively, $x1 = 2 * y1$ and $2 * x1 = y1$.

Expressing a condition along a path in terms of program input variables is known as *interpretation* of that condition. Obviously, the interpretation depends on the values of the program variables at the time the condition is evaluated.

Thus, the domain for path p_0 consists of all inputs for which $x1 = y1$. Similarly, the domains for the remaining two paths consist of all inputs that satisfy the corresponding conditions. Sample inputs in the domain of each of the three paths are given in the following as pairs of input integers.

$p_0 : (4, 4), (10, 10)$
$p_{11} : (4, 2), (8, 4)$
$p_{12} : (4, 8), (5, 10)$

Example 2.8 Consider a program that takes two inputs denoted by integer variables x and y. Now consider the following sequence of three conditions that must evaluate to true in order for a path in this program to be executed.

$y < x + 2$
$y > 1$
$x < 0$

The above conditions define a path domain shown in Figure 2.6. Each of the above three conditions leads to a *border segment*, or simply *border*, shown as a darkened line and marked with an arrow. The path domain is the shaded area bounded by the three borders. All values of x and y that lie inside the domain will cause the path to be executed. Points that lie outside the domain or at the boundary will cause at least one of the above conditions to be false and hence the path will not be executed. For example, point c in the figure corresponds to $(-0.5, 1)$. For this point the condition $y > 1$ is false.

Every path domain has a border along which lie useful tests.

The path conditions in the above examples are linear and hence lead to straight line borders. Also, here we have assumed that the inputs are numbers, integers, or floating point values. In general, path condition dimensions could be arbitrary. Let $relop = \{<, \leq, >, \geq, =, \neq\}$. A general

Figure 2.6 Pictorial representation of a path domain corresponding to the condition $y < x + 2 \wedge y > 1 \wedge x < 0$. The shaded triangular area represents the path domain which is a subdomain of the entire program domain. Three sample points labeled *a*, *b*, and *c*, are shown. Point *a* lies inside the path domain, *b* lies outside, and *c* lies on one of the borders.

form of a set of m path conditions in n inputs $x_1, x_2, \ldots, x_n$ can be expressed as follows.

$$\begin{array}{lll} f1(x_1, x_2, \ldots, x_n) & relop & 0 \\ f2(x_1, x_2, \ldots, x_n) & relop & 0 \\ \vdots & & \\ fm(x_1, x_2, \ldots, x_n) & relop & 0 \end{array}$$

In case all functions in the above set of conditions are linear, the path conditions can be written as follows.

$$\begin{array}{lll} a_{11}x_1 + a_{12}x_2, \ldots + a_{1n}x_n + c_1 & relop & 0 \\ a_{21}x_1 + a_{22}x_2, \ldots + a_{2n}x_n + c_2 & relop & 0 \\ \vdots & & \\ a_{m1}x_1 + a_{m2}x_2, \ldots + q_{mn}x_n + c_m & relop & 0 \end{array}$$

In the above conditions, a_{ij} denote the $m \times n$ coefficients of the input variables and c_j are m constants. Using the above standard notation, the path conditions in Example 2.8 can be expressed as follows.

$$\begin{array}{lll} -x + y - 2 & < & 0 \\ -y + 1 & < & 0 \\ x & < & 0 \end{array}$$

Figure 2.7 Non-linear path domain corresponding to the path condition $y > x^2 \wedge y < 1$. The arrows indicate the closed boundary of the parabolic domain.

In the above set of equations, the coefficients are given below.

$$\begin{array}{lll} a_{11} = -1 & a_{12} = 1 & c_1 = -2 \\ a_{21} = 0 & a_{22} = -1 & c_2 = 1 \\ a_{31} = -1 & a_{32} = 0 & c_3 = 0 \end{array}$$

Path conditions could also be non-linear. They could also contain arrays, strings, and other data types commonly found in programming languages. The next example illustrates a few other types of path conditions and the corresponding path domains.

Non-linear path conditions may lead to complex path domains thereby increasing the difficulty of finding a minimal set of tests.

Example 2.9 Consider the following two path conditions.

$$\begin{array}{lcl} y - x^2 & > & 0 \\ y & < & 1 \end{array}$$

Figure 2.7 is a graphical representation of the path domain that corresponds to the above conditions. The domain consists of points inside of a parabola bounded at the top by a line corresponding to the constraint $y < 1$.

Example 2.10 Often a condition in an `if` or a `while` loop contains reference to an array element. For instance, consider the following statements.

```
int x, n;
input (x, n);
inputArray(n, A);
....
```

```
5   if(A[i]>x){
6   ....
```

The input domain of the above program consists of triples (x, n, A). The constraint $A[i] > x$ defines a path domain consisting of all triples (x, n, A'), where A' is an array whose i^{th} element is greater than x. Of course, to define this domain more precisely one needs to know the range of variable i. Suppose that $0 \leq i \leq 10$. In that case the path domain consists of all arrays of length 0 through 10 at least one of whose elements satisfies the path condition.

Just as in the case of arrays, it is often challenging to define path domains for abstract data types that occur in OO languages such as Java and C++. Consider, for example, a stack as used in the following program segment.

```
1   Stack s;
2    if (full(s))
3        output("Stack is full");
4   ....
```

The path condition in the above program segment is *full*(s). Given stacks of arbitrary sizes, the path domain obviously consists of only stacks that are full. In situations like this, the program domain and the path domain are generally restricted for the purpose of testing by limiting stack size. Once done, one could consider stacks that are empty, full, or neither. Doing so partitions the program domain with respect to a stack into three subdomains one of which is a path domain.

That brings us to the end of this section to illustrate path conditions and path domains. Later in Chapter 4, we will see how to use path conditions to generate tests using a technique known as *domain testing*. (Also see Exercise 2.2.)

A domain error is said to exist in a program if for some input an incorrect path leads to an incorrect output.

2.2.6 Domain and computation errors

A computation error is said to exist in a program if for some input a correct path leads to an incorrect output.

Domain and computation errors are two error types based on the manner in which a program input leads to incorrect behavior. Specifically, a program P is said to contain a *domain* error if for a given input it follows a wrong path that leads to an incorrect output. P is said to contain a *computation* error if an input forces it to follow the correct path resulting in an incorrect output. An error in a predicate used in a conditional statement such as an `if` or a `while` leads to a domain error. An error in an

assignment statement that computes a variable used later in a predicate may also lead to a domain error.

A domain error could occur because an incorrect path has been selected in which case it is considered as a *path selection* error. It could also occur due to a missing path in which case it is known as a *missing path* error. We shall return to the notion of domain errors in Chapter 4 while discussing domain testing as a technique for test data generation.

2.2.7 Static code analysis tools and static testing

A variety of questions related to code behavior are likely to arise during the code inspection process described briefly in Chapter 1. Consider the following example: A code inspector asks "Variable `accel` is used at line 42 in module `updateAccel` but where is it defined?" The author might respond as "`accel` is defined in module `computeAccel`". However, a static analysis tool could give a complete list of modules and line numbers where each variable is defined and used. Such a tool with a good user interface could simply answer the question mentioned above.

Static testing requires only an examination of a program through processes such as code walkthrough. Static analysis of code is an aid to static testing.

Static code analysis tools can provide control flow and data flow information. The control flow information, presented in terms of a CFG, is helpful to the inspection team in that it allows the determination of the flow of control under different conditions. A CFG can be annotated with data flow information to make a data flow graph. For example, to each node of a CFG, one can append the list of variables defined and used. This information is valuable to the inspection team in understanding the code as well as pointing out possible defects. Note that a static analysis tool might itself be able to discover several data flow related defects.

A possible data flow path begins at the definition of a variable and ends at its use. A CFG can be annotated with data flow information, i.e., pointing to the node where a variable is defined and another node where this variable is used.

Several such commercial as well as open source tools are available. Purify from IBM Rational and Klockwork from Klockwork Inc. are two of the many commercially available tools for static analysis of C and Java programs. LAPSE (Lightweight Analysis for Program Security in Eclipse) is an open source tool for the analysis of Java programs.

Example 2.11 Consider the CFGs in Figure 2.8 each of which has been annotated with data flow information. In (a) variable `x` is defined in block 1 and used subsequently in blocks 3 and 4. However, the CFG clearly shows that the definition of `x` at block 1 is used at block 3 but not at block 5. In fact the definition of `x` at block 1 is considered *killed* due to its redefinition at block 4.

Is the redefinition of `x` at block 5 an error? The answer to this question depends on the function being computed by the code

Figure 2.8 Partial CFGs annotated with data flow information. `d(x)` **and** `u(x)`**, respectively, imply the definition and use of variable** `x` **in a block. (a) CFG that indicates a possible data flow error. (b) CFG with a data flow error.**

segment corresponding to the CFG. It is indeed possible that the redefinition at block 5 is erroneous. The inspection team must be able to answer this question with the help of the requirements and the static information obtained from an analysis tool.

Figure 2.8(b) indicates the use of variable `y` in block 3. If `y` is not defined along the path from Start to block 3, then there is a data flow error as a variable is used before it is defined. Several such errors can be detected by static analysis tools.

2.3 Execution History

An execution trace of a program is a sequence of program states or partial program states. An execution slice is a subsequence of an execution trace.

Execution history of a program, also known as *execution trace*, is an organized collection of information about various elements of a program during a given execution. An execution slice is an executable sub-sequence of execution history. There are several ways to represent an execution history. For example, one possible representation is the sequence in which the functions in a program are executed against a given test input. Another representation is the sequence in which program blocks are executed. Thus, one could construct a variety of representations of the execution history of a program against a test input. For a program written in an object-oriented language like Java, an execution history could also be represented as a sequence of objects and the corresponding methods accessed.

Example 2.12 Consider Program P2.1 and its control flow graph in Figure 2.2 on page 78. We are interested in finding the sequence in which the basic blocks are executed when P2.1 is executed with the test input $t_1 :< x = 2, y = 3 >$. A straightforward examination of Figure 2.2 reveals the following sequence: 1, 3, 4, 5, 6, 5, 6, 5, 6, 7, 9. This sequence of blocks represents an execution history of P2.1 against test t_1. Another test $t_2 :< x = 1, y = 0 >$ generates the following execution history expressed in terms of blocks: 1, 3, 4, 5, 7, 9.

An execution history may also include values of program variables. Obviously, the more the information in the execution history, the larger the space required to save it. What gets included or excluded from an execution history depends on its desired use and the space available for saving the history. For debugging a function, one might want to know the sequence of blocks executed as well as values of one or more variables used in the function. For selecting a subset of tests to run during regression testing, one might be satisfied with only a sequence of function calls or blocks executed. For performance analysis, one might be satisfied with a trace containing the sequence in which functions are executed. The trace is then used to compute the number of times each function is executed to assess its contribution to the total execution time of the program.

A complete execution history recorded from the start of a program's execution until its termination represents a single execution path through the program. However, in some cases, such as during debugging, one might be interested only in partial execution history where program elements, such as blocks or values of variables, are recorded along a potion of the complete path. This portion might, for example, start when control enters a function of interest and end when the control exits this function.

2.4 Dominators and Post-Dominators

Given nodes n and m in a CFG, n is said to dominate m if every path that leads to m must go through n. The dominator relation and its variants are useful in deriving test cases, and during static and dynamic analysis of programs.

Let $G = (N, E)$ be a control flow graph for program P. Recall that G has two special nodes labeled `Start` and `End` . We define the dominator and post-dominator as two relations on the set of nodes N. These relations find various applications, especially during the construction of tools for test adequacy assessment (Chapter 7) and regression testing (Chapter 9).

For nodes n and m in N, we say that n dominates m if n lies on every path from `Start` to m. We write $dom(n, m)$ when n dominates m. In an analogous manner, we say that node n post-dominates node m if n lies on every path from m to the `End` node. We write $pdom(n, m)$ when n

Figure 2.9 (a) Dominator and (b) post-dominator trees derived from the flow graph in Figure 2.4.

post-dominates m. When $n \neq m$, we refer to these as strict dominator and strict post-dominator relations. $Dom(n)$ and $PDom(n)$ denote, respectively the sets of dominators and post-dominators of node n.

For $n, m \in N$, n is the immediate dominator of m when n is the last dominator of m along a path from the `Start` to m. We write $idom(n, m)$ when n is the immediate dominator of m. Each node, except for `Start`, has a unique immediate dominator. Immediate dominators are useful in that we can use them to build a dominator tree. A dominator tree derived from G succinctly shows the dominator relation.

For $n, m \in N$, n is an immediate post-dominator of m if n is the first post-dominator along a path from m to `End` . Each node, except for `End`, has a unique immediate post-dominator. We write $ipdom(n, m)$ when n is the immediate post-dominator of m. Immediate post-dominators allow us to construct a post-dominator tree that exhibits the post-dominator relation amongst the nodes in G.

Example 2.13 Consider the flow graph in Figure 2.4. This flow graph contains six nodes including `Start` and `End`. Its dominator and post-dominator trees are shown in Figure 2.9(a) and (b), respectively. In the dominator tree, for example, a directed edge connects an immediate dominator to the node it dominates. Thus, among other relations, we have $idom(1, 2)$ and $idom(4, \texttt{end})$. Similarly, from the post-dominator tree, we obtain $ipdom(4, 2)$ and $ipdom(\texttt{End}, 4)$.

Given a dominator and a post-dominator tree, it is easy to derive the set of dominators and post-dominators for any node. For

example, the set of dominators for node 4, denoted as *Dom*(4) is {2, 1, `Start`}. *Dom*(4) is derived by first obtaining the immediate dominator of 4 which is 2, followed by the immediate dominator of 2 which is 1, and lastly the immediate dominator of 1, which is `Start`. Similarly, we can derive the set of post-dominators for node 2 as {4, `End`}.

2.5 Program Dependence Graph

A program dependence graph captures data and control dependence relations in a program.

A program dependence graph (PDG) for program P exhibits different kinds of dependencies amongst statements in P. For the purpose of testing, we consider data dependence and control dependence. These two dependencies are defined with respect to data and predicates in a program. Next we explain data and control dependence, how they are derived from a program, and their representation in the form of a PDG. We first show how to construct a PDG for programs with no procedures and then show how to handle programs with procedures.

2.5.1 Data dependence

Statements in a program exhibit a variety of dependencies. For example, consider Program P2.2. We say that the statement at line 8 depends on the statement at line 4 because line 8 may use the value of variable `product` defined at line 4. This form of dependence is known as *data dependence*.

Data dependence can be visually expressed in the form of a data dependence graph (DDG). A DDG for program P contains one unique node for each statement in P. Declaration statements are omitted when they do not lead to the initialization of variables. Each node in a DDG is labeled by the text of the statement as in a CFG or numbered corresponding to the program statement.

Two types of nodes are used: a predicate node is labeled by a predicate such as a condition in an `if` or a `while` statement and a data node labeled by an assignment, input, or output statement. A directed arc from node n_2 to n_1 indicates that node n_2 is data dependent on node n_1. This kind of data dependence is also known as flow dependence. A definition of data dependency follows.

Data dependence

Let D be a DDG with nodes n_1 and n_2. Node n_2 is data dependent on n_1 if (a) variable v is defined at n_1 and used at n_2 and (b) there exists a path for non-zewro length from n_1 to n_2 not containing any node that redefines v.

Example 2.14 Consider Program P2.2 and its data dependence graph in Figure 2.10. The graph shows seven nodes corresponding to the program statements. Data dependence is exhibited using directed edges. For example, node 8 is data dependent on nodes 4, 7, and itself because variable `product` is used at node 8 and defined at nodes 4 and 8, and variable `num` is used at node 8 and defined at node 7. Similarly, node 11 corresponding to the output statement, depends on nodes 4 and 8 because variable `product` is used at node 11 and defined at nodes 4 and 8.

Notice that the predicate node 6 is data dependent on nodes 5 and 9 because variable `done` is used at node 6 and defined through an input statement at nodes 5 and 9. We have omitted from the graph the declaration statements at lines 2 and 3 as the variables declared are defined in the input statements before use. To be complete, the data dependency graph could add nodes corresponding to these two declarations and add dependency edges to these nodes (see Exercise 2.7).

2.5.2 Control dependence

Another form of dependence is known as *control dependence*. For example, the statement at line 12 in Program P2.1 depends on the predicate at line 10. This is because control may or may not arrive at line 12 depending on the outcome of the predicate at line 10. Note that the statement at line 9 does not depend on the predicate at line 5 because control will arrive at line 9 regardless of the outcome of this predicate.

As with data dependence, control dependence can be visually represented as a control dependence graph (CDG). Each program statement corresponds to a unique node in the CDG. There is a directed edge from node n_2 to n_1 when n_2 is control dependent on n_1.

Figure 2.10 Data dependence graph for Program P2.2. Node numbers correspond to line numbers. Declarations have been omitted.

Control dependence

Let C be a CDG with nodes n_1 and n_2, n_1 being a predicate node. Node n_2 is control dependent on n_1 if there is at least one path from n_1 to program exit that includes n_2 and at least one path from n_1 to program exit that excludes n_2.

Example 2.15 Figure 2.11 shows the control dependence graph for P2.2. Control dependece edges are shown as dotted lines.

As shown, nodes 7, 8, and 9 are control dependent on node 6 because there exists a path from node 6 to each of the dependent nodes as well as a path that excludes these nodes. Notice that none of the remaining nodes is control dependent on node 6, the only predicate node in this example. Node 11 is not control dependent on node 6 because any path that goes from node 6 to program exit includes node 11.

Now that we know what data and control dependence is, we can show program dependence graph as a combination of the two dependences. Each program statement contains one node in the PDG. Nodes are joined with directed arcs showing data and control dependencies. A PDG can be considered as a combination of two subgraphs: a data dependence subgraph and a control dependence subgraph.

Example 2.16 Figure 2.12 shows the program dependence graph for P2.2. It is obtained by combining the graphs shown in Figures 2.10 and 2.11 (see Exercise 2.7).

2.5.3 Call graph

As the name implies, a call graph is a directed graph in which each node is a call site, e.g., a procedure, and each edge connects two nodes (f, g) indicating that procedure f calls procedure g. For an object-oriented language such as C# or Java, a call graph captures calls to methods across objects.

A call graph captures the call-relation among functions (or methods) in a program.

Figure 2.11 Control dependence graph for Program P2.2.

Figure 2.12 Program dependence graph for P2.2.

A static call graph represents all potential procedure calls. A dynamic call graph captures calls that were actually made during one or more executions of the program. An exact static call graph is one that captures all potential call relationships. The problem of generating an exact static call graph is undecidable.

2.6 Strings, Languages, and Regular Expressions

Strings play an important role in testing. As we shall see in Section 5.2, strings serve as inputs to a finite state machine and hence to its implementation as a program. Thus, a string serves as a test input. A collection of strings also forms a language . For example, a set of all strings consisting of zeros and ones is the language of binary numbers. In this section we provide a brief introduction to strings and languages.

An alphabet is a set of symbols. A string over an alphabet is any sequence of symbols in the alphabet. A language, finite or infinite, is a set of strings over an alphabet.

A collection of symbols is known as an *alphabet*. We will use an upper case letter such as X and Y to denote alphabets. Though alphabets can be infinite, we are concerned only with finite alphabets. For example, $X = \{0, 1\}$ is an alphabet consisting of two symbols 0 and 1. Another alphabet is $Y = \{\text{dog}, \text{cat}, \text{horse}, \text{lion}\}$ that consists of four symbols "dog", "cat", "horse", and "lion".

A string *over* an alphabet X is any sequence of zero or more symbols that belong to X. For example, 0110 is a string over the alphabet {0, 1}. Also, `dog cat dog dog lion` is a string over the alphabet $\{dog, cat, horse, lion\}$. We will use lower case letters such as p, q, r to denote strings. The length of a string is the number of symbols in that string. Given a string s, we denote its length by $|s|$. Thus, $|1011| = 4$ and $|\texttt{dog cat dog}| = 3$. A string of length 0, also known as an *empty string*, is denoted by ϵ.

Let s_1 and s_2 be two strings over alphabet X. We write $s_1 \bullet s_2$ to denote the *concatenation* of strings s_1 and s_2. For example, given the alphabet $X = \{0, 1\}$, and two strings 011 and 101 over X, we obtain $011 \bullet 101 = 011101$. It is easy to see that $|s_1 \bullet s_2| = |s_1| + |s_2|$. Also, for any string s, we have $s \bullet \epsilon = s$ and $\epsilon \bullet s = s$.

A set L of strings over an alphabet X is known as a *language*. A language can be finite or infinite. Given languages L_1 and L_2, we denote their catenation as $L_1 \bullet L_2$ that denotes the set L defined as:

$$L = L_1 \bullet L_2 = \{x \bullet y | x \in L_1, y \in L_2\}$$

The following sets are finite languages over the binary alphabet {0, 1}.

∅: The empty set

{∈}: A language consisting only of one string of length zero

{00, 11, 0101}: A language containing three strings

A regular expression is a convenient means for a compact representation of sets of strings. For example, the regular expression (01)* denotes the set of strings that consists of the empty string, the string 01, and all strings obtaining by concatenating the string 01 with itself one or more times. Note that (01)* denotes an infinite set. A formal definition of regular expressions follows.

Regular Expressions
Given a finite alphabet X, the following are regular expressions over X:

- If a belongs to X, then a is a regular expression that denotes the set $\{a\}$.
- Let r_1 and r_2 be regular expressions over the alphabet X that denote, respectively, sets L_1 and L_2. Then $r_1 \bullet r_2$ is a regular expression that denotes the set $L_1 \bullet L_2$.
- If r is a regular expression that denotes the set L, then r^+ is a regular expression that denotes the set obtained by concatenating L with itself one or more times also written as L^+. Also, r^*, known as the *Kleene closure* of r, is a regular expression. If r denotes the set L then r^* denotes the set $\{\epsilon\} \cup L^+$.
- If r_1 and r_2 are regular expressions that denote, respectively, sets L_1 and L_2, then $r_1|r_2$ is also a regular expression that denotes the set $L_1 \cup L_2$.

Regular expressions are useful in expressing both finite and infinite test sequences. For example, if a program takes a sequence of zeroes and ones and flips each zero to a 1 and a 1 to a 0, then a few possible sets of test inputs are 0^*, $(10)^+$, 010|100.

2.7 Tools

A large variety of tools are available for static analysis of code in various languages. Below is a sample of such tools.

```
Tool: EclipseCFG
Link: http://eclipsefcg.sourceforge.net/
Description
```

EclipseCFG is an Eclipse plugin for generating CFG for Java code. The CFG is generated for each method in the selected class. The CodeCover tool is integrated with Eclipse and EclipseCFG. This integration enables the CFG to graphically indicate statement and condition coverage. The tool also computes the McCabe complexity metric for each method.

```
Tool: LDRS Testbed.
Link: http://www.ldra.com/index.php/en/products-a-
  services/ldra-tool-suite/ldra-testbedr
Description
```

This is a commercial-strength static and dynamic analysis tool set. It analyzes the source code statically by doing a complete syntax analysis either for individual functions or for the entire system. The tool provides statement, branch, and MC/DC coverage. Hence, this tool is useful in environments that develop code that must meet DO-178B certification. The coverage results are displayed in a tabular form and can also be viewed in the form of a dynamic call graph.

```
Tool: CodeSonar
Link: http://www.grammatech.com/products/codesonar/
  support.html
Description
```

CodeSonar is a static analysis tool for C and C++ programs. It attempts to detect several problems in the code including data races, deadlocks, buffer overruns, and memory usage after it has been freed. It also includes a powerful feature for application architecture visualization.

```
Tool: gprof
Link: http://sourceware.org/binutils/docs/gprof/
Description
```

gprof is a profiler to collect run time statistics of a program. It generates a dynamic call graph of a program. The call graph is presented in textual form.

SUMMARY

In this chapter, we have presented some basic mathematical concepts likely to be encountered by any tester. Control flow graphs represent the flow of control in a program. These graphs are used in program analysis. Many test tools transform a program into its control flow graph for the purpose of analysis. Section 2.2 explains what a control flow graph is and how to construct one. Basis paths are introduced here. Later in Chapter 4, basis paths are used to derive test cases.

Dominators and program dependence graphs are useful in static analysis of programs. Chapter 9 shows how to use dominators for test minimization.

An finite state machine recognizes strings that can be generated using regular expressions. Section 2.6 covers strings and languages useful in understanding the model based testing concepts presented in Chapter 5.

Exercises

2.1 (a) Calculate the length of the path traversed when P2.2 is executed on an input sequence containing N integers. (b) Suppose that the statement on line 8 of P2.2 is replaced by the following.

```
8    if(num>0)product=product*num;
```

Calculate the number of distinct paths in the modified P2.2 if the length of the input sequence can be 0, 1, or 2. (c) For an input sequence of length N, what is the maximum length of the path traversed by the modified P2.2?

2.2 (a) List the different paths in the program in Example 2.7 when the loop body is traversed twice. (b) For each path list the path conditions and a few sample values in the corresponding input domain.

2.3 Find the basis set and the corresponding path vectors for the flow graph in Figure 2.2. Construct a path p through this flow graph that is not a basis path. Find the path vector V_p for p. Express V_p as a linear combination of the path vectors for the basis paths.

2.4 Solve Exercise 2.7 but for the program in Example 2.7.

2.5 Let P denote a program that contains a total of n simple conditions in `if` and `while` statements. What is the maximum number of elements in the basis set for P? What is the largest path vector in P?

2.6 The basis set for a program is constructed using its CFG. Now suppose that the program contains compound predicates. Do you think the basis set for this program will depend on how compound predicates are represented in the flow graph? Explain your answer.

Figure 2.13 Control flow graphs for Exercise 2.7.

2.7 Construct the dominator and post-dominator trees for the control flow graph in Figure 2.2 and for each of the control flow graphs in Figure 2.13.

2.8 Let `pred`(n) be the set of all predecessors of node n in a CFG $G = (N, E)$. The set of dominators of n can be computed using the following equations:

$$Dom(\texttt{Start}) = \{\texttt{Start}\}$$

$$Dom(n) = \{n\} \cup \{ \bigcap_{p \in \texttt{pred}(\mathrm{n})} Dom(p)\}$$

Using the above equation, develop an algorithm to compute the dominators of each node in N. (*Note: There exist several algorithms for computing the dominators and post-dominators in a CFG. Try developing your own and then study the algorithms in relevant citations in the Bibliography.*)

2.9 (a) Modify the dependence graph in Figure 2.10 by adding nodes corresponding to the declarations at lines 2 and 3 in Program P2.2 and the corresponding dependence edges. (b) Can you offer a reason why for Program P2.2 the addition of nodes corresponding to the declarations is redundant? (c) Under what condition would it be useful to add nodes corresponding to declarations to the data dependence graph?

2.10 Construct a program dependence graph for the exponentiation program P2.1.

2.11 What is the implication of a cycle in a call graph?

Part II
Test Generation

Generation of test inputs and the corresponding expected outputs is an essential activity in any test organization. The input data and the corresponding expected output are wedded into a test case. A collection of test data becomes a test set or a test suite.

There exist a wide range of guidelines, techniques, and supporting tools to generate test cases. On one extreme, these include guidelines such as the use of boundary value analysis while on the other there exist formal techniques for generating test data from formal specifications such as those specified in the Z notation. In addition, there are test generation techniques that rely exclusively on the code under test and generate tests that satisfy some code coverage criteria.

Chapters in this part of the book bring to you a varied collection of guidelines and techniques for the generation of test sets. While some techniques presented herein are basic and widely applicable, such as equivalence partitioning, boundary value analysis, and random testing, others such as test generation from finite state machines find uses in specific domains.

This page is intentionally left blank.

3

Domain Partitioning

CONTENTS

3.1 Introduction
3.2 The test selection problem
3.3 Equivalence partitioning
3.4 Boundary value analysis
3.5 Category-partition method

The purpose of this chapter is to introduce techniques for the generation of test data from informally and formally specified requirements. Essentially, conditions in the form of predicates are extracted either from the informally or formally specified requirements or directly from the program under test. There exists a variety of techniques to use one or more predicates as input and generate tests. Some of these techniques can be automated while others may require significant manual effort for large applications. It is not possible to categorize the techniques in this chapter as either black- or white-box technique. Depending on how tests are generated, a technique could be termed as a black- or white-box technique.

3.1 Introduction

Requirements may be formal or informal. In either case, they serve as a key source of tests.

Requirements serve as the starting point for the generation of tests. During the initial phases of development, requirements may exist only in the minds of one or more people. These requirements, more aptly ideas, are then specified rigorously using modeling elements such as use cases, sequence diagrams, and statecharts in UML. Rigorously specified requirements are often transformed into formal requirements using requirements specification languages such as Z, S, and RSML.

While a complete formal specification is a useful document, it is often the case that aspects of requirements are captured using appropriate modeling formalisms. For example, Petri nets and its variants are used for specifying timing and concurrency properties in a distributed system, timed input automata to specify timing constraints in a real-time system, and finite state machines to capture state transitions in a protocol. UML is an interesting formalism in that it combines into a single framework several different notations used for rigorously and formally specifying requirements.

A requirement specification can thus be informal, rigorous, formal, or a mix of these three approaches. Usually, requirements of commercial applications are specified using a mix of the three approaches. In either case, it is the task of the tester, or a test team, to generate tests for the entire application regardless of the formalism in which the specifications are available. The more formal the specification, the higher are the chances of automating the test generation process. For example, specifications using finite state machines, timed automata, and Petri nets can be the inputs to a programmed test generator and tests generated automatically. However, significant manual effort is required when generating test cases from use cases.

Often, high-level designs are also considered as part of requirement specification. For example, high-level sequence and activity diagrams in UML are used for specifying interaction amongst high-level objects. Tests can also be generated from such high-level designs.

In this chapter, the focus is on the generation of tests from informal and rigorously specified requirements. These requirements serve as a source for the identification of the input domain of the application to be developed. A variety of test generation techniques are available to select a subset of the input domain to serve as test set against which the application will be tested.

Figure 3.1 lists the techniques described in this chapter. The figure shows requirements specification in three forms: informal, rigorous, and formal. The input domain is derived from the informal and rigorous specifications. The input domain then serves as a source for test selection. Various

Figure 3.1 Techniques listed here, and marked with an asterisk, for test selection from informal and rigorously specified requirements, are introduced in this chapter. Techniques from formally specified requirements using graphical models and logic-based languages are discussed in other chapters.

techniques, listed in the figure, are used to select a relatively small number of test cases from a usually large input domain.

The remainder of this chapter describes each of the techniques marked in the figure. All techniques described here fall under the "black-box" testing category. However, some could be enhanced if the program code is available. Such enhancements are discussed in Part III of this book.

The set of all possible inputs to a program is almost always extremely large. The test selection problem is to find a small subset of all possible inputs, and the corresponding expected outputs, that will serve as a test suite.

3.2 The Test Selection Problem

Let D denote the input domain of program p. The test selection problem is to select a subset T of tests such that execution of p against each element of T will reveal all errors in p. In general there does not exist any algorithm to construct such a test set. However, there are heuristics and model-based

An input domain of a program can be defined in at least two ways. According to one definition, it is the set of all possible legal inputs to a program. According to another, it is the set of all possible inputs to a program.

methods that can be used to generate tests that will reveal certain types of faults. The challenge is to construct a test set $T \subseteq D$ that will reveal as many errors in p as possible. As discussed next, the problem of test selection is difficult due primarily to the size and complexity of the input domain of p.

An input domain for a program is the set of all possible *legal* inputs that the program may receive during execution. The set of legal inputs is derived from the requirements. In most practical problems, the input domain is large, i.e. has many elements, and often complex, i.e. the elements are of different types such as integers, strings, reals, Boolean, and structures.

The large size of the input domain prevents a tester from exhaustively testing the program under test against all possible inputs. By "exhaustive" testing, we mean testing the given program against every element in its input domain. The complexity makes it harder to select individual tests. The following two examples illustrate what causes large and complex input domains.

Example 3.1 Consider program P that is required to sort a sequence of integers into ascending order. Assuming that P will be executed on a machine in which integers range from -32768 to 32767, the input domain of P consists of all possible sequences of integers in the range $[-32768, 32767]$.

If there is no limit on the size of the sequence that can be an input, then the input domain of P is infinitely large and P can never be tested exhaustively. If the size of the input sequence is limited to, say, $N > 1$, then the size of the input domain depends on the value of N. In general, denoting the size of the input domain by S, we get

$$S = \Sigma_{i=0}^{N} v^i,$$

where v is the number of possible values each element of the input sequence may assume, which is 65536. Using the formula given above, it is easy to verify that the size of the input domain for this simple sort program is enormously large even for small values of N. Even for small values of N, say $N = 3$, the size of the input space is large enough to prevent exhaustive testing.

Example 3.2 Consider a procedure P in a payroll processing system that takes an employee record as input and computes the weekly salary. For simplicity, assume that the employee record consists of the following items with their respective types and constraints:

ID: `int`;	`ID` is 3-digits long from 001 to 999.
name: `string`;	`name` is 20 characters long; each character belongs to the set of 26 letters and a space character.
rate: `float`;	`rate` varies from \$5 to \$10 per hour; rates are in multiples of a quarter.
hoursWorked: `int`;	`hoursWorked` varies from 0 to 60.

Each element of the input domain of P is a structure that consists of four items listed above. This domain is large as well as complex. Given that there are 999 possible values of ID, 27^{20} possible character strings representing names, 21 hourly pay rates, and 61 possible work hours, the number of possible records is

$$999 \times 27^{20} \times 21 \times 61 \approx 5.42 \times 10^{34}.$$

Once again, we have a huge input domain that prevents exhaustive testing. Notice that the input domain is large primarily due to the large number of possible names that represent strings and the combinations of the values of different fields.

For most useful software, the input domains are even larger than the ones in the above examples. Further, in several cases, it is often difficult to even characterize the complete input domain due to relationships between the inputs and timing constraints. Thus, there is a need to use methods that will allow a tester to be able to select a much smaller subset of the input domain for the purpose of testing a given program. Of course, as we will learn in this book, each test selection method has its strengths and weaknesses.

3.3 Equivalence Partitioning

Equivalence partitioning is used to partition an input domain into a relatively small number of subsets. Each subset, known as an equivalence class, is used as a source of a few tests.

Test selection using equivalence partitioning allows a tester to subdivide the input domain into a relatively small number of subdomains, say $N > 1$, as shown in Figure 3.2(a). In strict mathematical terms, the subdomains by definition are disjoint. The four subsets shown in Figure 3.2(a) constitute a partition of the input domain while the subsets in Figure 3.2(b) are not. Each subset is known as an *equivalence class*.

The equivalence classes are created assuming that the program under test exhibits the same behavior on all elements, i.e. tests, within a class. This assumption allows a tester to select exactly one test from each equivalence class resulting in a test suite of exactly N tests.

An equivalence class is a subset of the input domain. It consists of inputs on which the program under test is expected to behave in the same manner according to some well-defined criterion.

Figure 3.2 Four equivalence classes created from an input domain. (a) Equivalence classes together constitute a partition as they are disjoint. (b) Equivalence classes are not disjoint and do not form a partition. 1-2 indicates the region where subsets 1 and 2 overlap. This region contains test inputs that belong to subsets 1 and 2.

Of course, there are many ways to partition an input domain into equivalence classes. Hence, the set of tests generated using the equivalence partitioning technique is not unique. Even when the equivalence classes, created by two testers, are identical, the testers might select different tests. The fault detection effectiveness of the tests so derived depends on the tester's experience in test generation, familiarity with the requirements, and when the code is available, familiarity with the code itself. In most situations, equivalence partitioning is used as one of the several test generation techniques.

3.3.1 Faults targeted

It is important to test a program for both legal and illegal (or unexpected) inputs. Doing the latter aims at testing for the robustness of a program.

The entire set of inputs to any application can be divided into at least two subsets: one containing all expected (E), or legal, inputs and the other containing all unexpected (U), or illegal, inputs. Each of the two subsets, E and U, can be further subdivided into subsets on which the application is required to behave differently. Equivalence class partitioning selects tests that target any faults in the application that cause it to behave incorrectly when the input is in either of the two classes or their subsets. Figure 3.3 shows a sample subdivision of all possible inputs to an application.

For example, consider an application A that takes an integer denoted by *age* as input. Let us suppose that the only legal values of *age* are in the range [1, 120]. This set of input values is now divided into a set E containing all integers in the range [1, 120] and a set U containing the remaining integers.

Furthermore, assume that the application is required to process all values in the range [1, 61] in accordance with requirement R_1 and those in

Figure 3.3 Set of inputs partitioned into two regions one containing expected (*E*), or legal, and the other containing unexpected (*U*), or illegal, inputs. Regions *E* and *U* are further subdivided based on the expected behavior of the application under test. Representative tests, one from each region, is targeted at exposing faults that cause the application to behave incorrectly in their respective regions.

the range [62, 120] according to requirement R_2. Thus, E is further subdivided into two regions depending on the expected behavior. Similarly, it is expected that all invalid inputs less than 1 are to be treated in one way while all greater than 120 are to be treated differently. This leads to a subdivision of U into two categories.

To reduce the number of tests, inputs from an equivalence class are assumed to target the same kind of faults in a program. Under this assumption it is sufficient to select one test from each equivalence class.

Tests selected using the equivalence partitioning technique aim at targeting faults in A with respect to inputs in any of the four regions, i.e. two regions containing expected inputs and two regions containing the unexpected inputs. It is expected that any single test selected from the range [1, 61] will reveal any fault with respect to R_1. Similarly, any test selected from the region [62, 120] will reveal any fault with respect to R_2. A similar expectation applies to the two regions containing the unexpected inputs.

The effectiveness of tests generated using equivalence partitioning for testing application A is judged by the ratio of the number of faults these tests are able to expose to the total faults lurking in A. As is the case with any test selection technique in software testing, the effectiveness of tests selected using equivalence partitioning is less than 1 for most practical applications. However, the effectiveness can be improved through an unambiguous and complete specification of the requirements and carefully selected tests using the equivalence partitioning technique described in the following sections.

3.3.2 Relations

Recalling from our knowledge of sets, a relation is a set of n-ary tuples. For example, a method `addList` that returns the sum of elements in a list of integers defines a binary relation. Each pair in this relation consists of a

list and an integer that denotes the sum of all elements in the list. Sample pairs in this relation include ((1, 5), 6) and ((−3, 14, 3), 14), and (( ), 0). Formally, the relation computed by `addList` is defined as follows.

$$\texttt{addList}: \mathcal{L} \rightarrow \mathbb{Z}$$

A relation is a set of tuples. A function in a program defines a relation among its inputs and outputs. A function also defines a relation among its inputs.

where $\mathcal{L}$ is the set of all lists of integers and $\mathbb{Z}$ denotes the set of integers. Taking the example from the above paragraph as a clue, we can argue that every program, or method, defines a relation. Indeed, this is true provided the domain, i.e. the input set, and the range, i.e. the output set, are defined properly.

For example, suppose that the `addList` method has an error due to which it crashes when supplied with an empty list. In this case, even though `addList` is required to compute the relation defined in the above paragraph, it does not do so due to the error. However, it does compute the following relation:

$$\texttt{addList}: \mathcal{L} \rightarrow \mathbb{Z} \cup \{error\}$$

Relations that help a tester partition the input domain of a program are usually of the kind

$$R : \mathcal{I} \rightarrow \mathcal{I},$$

where $\mathcal{I}$ denotes the input domain. Relation R is *on* the input domain. It defines an *equivalence class* that is a subset of $\mathcal{I}$. The following examples illustrate several ways of defining R on the input domain.

Example 3.3 Consider a method `gPrice` that takes the name of a grocery item as input, consults a database of prices, and returns the unit price of this item. If the item is not found in the database, it returns with a message *Price information not available*.

The input domain of `gPrice` consists of names of grocery items which are of type `string`. Milk, Tomato, Yogurt, and Cauliflower are sample elements of the input domain. Obviously, there are many others. For this example, we assume that the price database is accessed through another method and hence is not included in the input domain of `gPrice`. We now define the following relation on the input domain of `gPrice`.

$$pFound : \mathcal{I} \rightarrow \mathcal{I}$$

The *pFound* relation associates elements t_1 and t_2 if `gPrice` returns a unit price for each of these two inputs. It also associates t_3 and t_4 if `gPrice` returns an error message on each of the two inputs. Now suppose that the price database is as in the following table.

Item	Unit Price
Milk	2.99
Tomato	0.99
Kellogg's Cornflakes	3.99

The inputs Milk, Tomato, and Kellogg's Cornflakes are related to each other through the relation *pFound*. For any of these inputs, `gPrice` is required to return the unit price. However, for input Cauliflower, and all other strings representing the name of a grocery item not on the list above, `gPrice` is required to return an error message. Arbitrarily constructed strings that do not name any grocery item belong to another equivalence class defined by *pFound*.

Any item that belongs to the database can be considered as a representative of the equivalence class. For example, Milk is a representative of one equivalence class denoted as [Milk] and [Cauliflower] is a representative of another equivalence class.

Relation *pFound* defines equivalence classes, say pF and pNF, respectively. Each of these classes is a subset of the input domain $\mathcal{I}$ of `gPrice`. Together, these equivalence classes form a partition of the input domain $\mathcal{I}$ because $pF \cup pNF = \mathcal{I}$ and $pF \cap pNF = \emptyset$.

In the previous example, the input assumes discrete values such as Milk and Tomato. Further, we assumed that `gPrice` behaves the same for all valid values of the input. There may be situations where the behavior of the method under test depends on the value of the input that could fall into any of several categories, most of them being valid. The next example shows how, in such cases, multiple relations can be defined to construct equivalence classes.

In several cases, multiple relations are useful in defining equivalence classes. Such cases arise when the program under test behaves differently for different classes of inputs.

Example 3.4 Consider an automatic printer testing application named `pTest`. The application takes the manufacturer and model of a printer as input and selects a test script from a list. The script is then executed to test the printer. Our goal is to test if the script selection part of the application is implemented correctly.

The input domain $\mathcal{I}$ of `pTest` consists of strings representing the printer manufacturer and model. If `pTest` allows textual input through keyboard, then strings that do not represent any printer

recognizable by `pTest` also belong to $\mathcal{I}$. However, if `pTest` provides a graphical user interface to select a printer, then $\mathcal{I}$ consists exactly of the strings offered in the pulldown menu.

The script selected depends on the type of the printer. For simplicity, we assume that the following three types are considered: color inkjet (ci), color laserjet (cl), and color multifunction (cm). Thus, for example, if the input to `pTest` is "HP Deskjet 6840" the script selected will be the one to test a color inkjet printer. The input domain of `pTest` consists of all possible strings, representing valid and invalid printer names. A valid printer name is one that exists in the database of printers used by `pTest` for the selection of scripts while an invalid printer name is a string that does not exist in the database.

For this example, we define the following four relations on the input domain. The first three relations below correspond to the three printer categories while the fourth relation corresponds to an invalid printer name.

$ci : \mathcal{I} \rightarrow \mathcal{I}$
$cl : \mathcal{I} \rightarrow \mathcal{I}$
$cm : \mathcal{I} \rightarrow \mathcal{I}$
$invP : \mathcal{I} \rightarrow \mathcal{I}$

Each of the four relations above defines an equivalence class. For example, relation *cl* places all color laserjet printers into one equivalence class and all other printers into another equivalence class. Thus, each of the four relations defines two equivalence classes for a total of eight equivalence classes. While each relation partitions the input domain of `pTest` into two equivalence classes, the eight classes overlap. Notice that relation *invP* might be empty if `pTest` offers a list of printers to select from.

We can simplify our task of partitioning the input domain by defining a single equivalence relation *pCat* that partitions the input domain of `pTest` into four equivalence classes corresponding to the categories *ci*, *cl*, *cm*, and *invP*.

In software testing, an equivalence partition is not always a true partition in a mathematical sense. This is because the equivalence classes may overlap.

The example above shows that equivalence classes may not always form a partition of the input domain due to overlap. The implication of such an overlap on test generation is illustrated in the next subsection.

The two examples above show how to define equivalence classes based on the knowledge of requirements. In some cases, the tester has access to both the program requirements and its code. The next example shows a few

ways to define equivalence classes based on the knowledge of requirements and the program text.

Example 3.5 The `wordCount` method takes a word *w* and a file name *f* as input and returns the number of occurrences of *w* in the text contained in the file named *f*. An exception is raised if there is no file with name *f*. Using the partitioning method described in the examples above, we obtain the following equivalence classes.

E1: Consists of pairs (*w*,*f*) where *w* is a string and *f* denotes a file that exists.
E2: Consists of pairs (*w*,*f*) where *w* is a string and *f* denotes a file that does not exist.

Now suppose that the tester has access to the code for `wordCount`. The partial pseudo code for `wordCount` is given below.

```
Program P3.1
1   begin
2     string w, f;
3     input (w, f);
4     if ( not exists(f)) {raise exception; return(0)};
5     if (length(w)==0)return(0);
6     if(empty(f)) return(0);
7     return(getCount(w, f));
8   end
```

The code above contains eight distinct paths created by the presence of the three `if` statements. However, as each `if` statement could terminate the program, there are only six feasible paths. We can define a relation named *covers* that partitions the input domain of `wordCount` into six equivalence classes depending on which of the six paths is covered by a test case. These six equivalence classes are defined in the following table.

All inputs that force a program to follow the same path, can be considered as constituting an equivalence class.

Equivalence class	w	f
E1	non-null	exists, non-empty
E2	non-null	does not exist
E3	non-null	exists, empty
E4	null	exists, non-empty
E5	null	does not exist
E6	null	exists, empty

> We note that the number of equivalence classes without any knowledge of the program code is two, whereas the number of equivalence classes derived with the knowledge of partial code is 6. Of course, an experienced tester will likely derive the six equivalence classes given above, and perhaps more, even before the code is available (see Exercise 3.6).

In each of the examples above, we focused on the inputs to derive the equivalence relations and, hence, equivalence classes. In some cases, the equivalence classes are based on the output generated by the program. For example, suppose that a program outputs an integer. It is worth asking: "Does the program ever generate a 0? What are the maximum and minimum possible values of the output?" These two questions lead to the following equivalence classes based on outputs.

E1: Output value *v* is 0.
E2: Output value *v* is the maximum possible.
E3: Output value *v* is the minimum possible.
E4: All other output values.

Based on the output equivalence classes, one may now derive equivalence classes for the inputs. Thus, each of the four classes given above might lead to one equivalence class consisting of inputs. Of course, one needs to carry out such output-based analysis only if there is sufficient reason to believe that it will likely generate equivalence classes in the input domain that cannot be generated by analyzing the inputs and program requirements.

3.3.3 Equivalence classes for variables

Tables 3.1 and 3.2 offer guidelines for partitioning variables into equivalence classes based on their type. The examples given in the tables are assumed to be derived from the application requirements. As explained in the following section, these guidelines are used while deriving equivalence classes from the input domain of an entire application that uses several input variables. Below we discuss the entries listed in Tables 3.1 and 3.2.

Each independent variable in a program can be used as a source of equivalence classes. The number and type of these classes depend on the type of the variable and the values it may assume.

Range: A range may be specified implicitly or explicitly. For example, the value of *speed* is in the explicitly specified range [60, 90], whereas the range for the values of *area* is specified implicitly. In the former case, one can identify values outside of the range. In the case of *area*, even though it is possible to identify values outside of the range, it may be impossible to input these values to the application because of representational

Table 3.1 Guidelines for generating equivalence classes for variables: range and strings.

Kind	Equivalence classes	Example	
		Constraint	**Class representatives**†
Range	One class with values inside the range and two with values outside the range.	*speed* $\in$ [60, 90]	{{50}↓, {75}↑, {92}↓}
		area : `float`; *area* ≥ 0	{{−1.0}↓, {15.52}↑}
		age : `int`; $0 \leq$ *age* $\leq$ 120	{{−1}↓, {56}↑,{132}↓}
		letter : `char`;	{{J}↑, {3}↓}
String	At least one containing all legal strings and one containing all illegal strings. Legality is determined based on constraints on the length and other semantic features of the string.	*fname* : `string`;	{{ϵ}↓, {Sue}↑, {Sue2}↓, {Too Long a name}↓}
		vname : `string`;	{{ϵ}↓, {shape}↑, {address1}↑}, {Long variable }↓}

†Symbols following each equivalence class: ↓: Representative from an equivalence class containing illegal inputs. ↑: Representative from an equivalence class containing legal inputs.

In some cases, determining the range of values of a variable could require the examination of program code. This could happen when the requirements are specified informally.

constraints imposed by the underlying hardware and software of the machine on which the application is being tested.

The range for the values of *age* has also been specified implicitly. However, if, for example, *age* represents the age of an employee in a payroll processing application, then the tester must be able to identify the range of values from the knowledge of the semantics of *age*. In this case, *age* cannot be less than 0 and 120 is a reasonable upper limit. The equivalence classes

Table 3.2 Guidelines for generating equivalence classes for variables: enumeration and arrays.

Kind	Equivalence classes	Example†	
		Constraint	Class representatives‡
Enumeration	Each value in a separate class.	*auto_color* ∈ {`red`, `blue`, `green`}	{{`red`}↑, {`blue`}↑, {`green`}↑}
		up: `boolean`	{{`true`}↑, {`false`}↑}
Array	One class containing all legal arrays, one containing only the empty array, and one containing arrays larger than the expected size.	Java array: `int[]` *aName* = `new int`[3];	{{[ ]}↓, {[−10, 20]}↑, {[−9, 0, 12, 15]}↓}

†See text for an explanation of various entries.
‡Symbols following each equivalence class: ↓: Representative from an equivalence class containing illegal inputs. ↑: Representative from an equivalence class containing legal inputs.

for *letter* have been determined based on the assumption that *letter* must be one of 26 letters A through Z.

While a variable might take on values from within a range, this range might actually be divided into several sub-ranges. Each such sub-range leads to multiple equivalence classes.

In some cases, an application might subdivide an input into multiple ranges. For example, in an application related to social security, *age* may be processed differently depending on which of the following four ranges it falls into: [1, 61], [62, 65], [67, 67], and [67, 120]. In this case, there is one equivalence class for each range and one each for a values less than the smallest value and larger than the highest value in any range. For this example, we obtain the following representatives from each of the six equivalence classes: 0 (↓), 34 (↑), 64 (↑), 67 (↑), 72 (↑), and 121 (↓). Further, if there is reason to believe that 66 is a special value and will be processed in some special way by the application, then 66 (↓) must be placed in a separate equivalence class, where, as before, ↓ and ↑ denote, respectively, values from equivalence classes containing illegal and legal inputs.

Strings: Semantic information has also been used in the table to partition values of *fname*, which denotes the first name of a person, and *vname*, which denotes the name of a program variable. We have assumed that the first name must be a nonempty string of length at most 10 and must contain only alphabetic characters; numerics and other characters are not allowed. Hence ϵ, denoting an empty string, is an invalid value in its own equivalence class, while *Sue*29 and *Too Long a name* are two more invalid values in their own equivalence classes. Each of these three invalid values have been placed in their own class as they have been derived from different semantic constraints. Similarly, valid and invalid values have been selected for *vname*. Note that while the value *address*1, which contains a digit, is a legal value for *vname*, *Sue*2, which also contains a digit, is illegal for *fname*.

One obvious equivalence class for variables of type string contains only the empty string, while the other contains all the non-empty strings. However, a knowledge of the requirements might be used to impose constraints on the maximum length of strings as well as on the actual strings themselves and thus arrive at more than two equivalence classes.

Enumerations: The third set of rows in Table 3.1 offers guidelines for partitioning a variable of enumerated type. If there is a reason to believe that the application is to behave differently for each value of the variable, then each value belongs to its own equivalence class. This is most likely to be true for a Boolean variable. However, in some cases, this may not be true. For example, suppose that *auto_color* may assume three different values as shown in the table, but will be processed only for printing. In this case, it is safe to assume that the application will behave the same for all possible values of *auto_color* and only one representative needs to be selected. When there is reason to believe that the application does behave differently for different values of *auto_color*, then each value must be placed in a different equivalence class as shown in the table.

Each member of an enumeration belongs to its own equivalence class.

In the case of enumerated types, as in the case of some range specifications, it might not be possible to specify an illegal test input. For example, if the input variable *up* is of type Boolean and can assume both `true` and `false` as legal values, then all possible equivalence classes for *up* contain legal values.

Arrays: An array specifies the size and type of each element. Both of these parameters are used in deriving equivalence classes. In the example shown, an array may contain at least one and at most three elements. Hence, the empty array and the one containing four elements are the illegal inputs. In the absence of any additional information, we do not have any constraint on the elements of the array. In case we do have any such constraint, then additional classes could be derived. For example, if each element of the array must be in the range [−3, 3], then each of the values specified in the example is illegal. A representative of the equivalence class containing legal values is [2, 0, 3].

Equivalence classes for variables of type array can be derived by imposing constraints on the length of the array and on the values of its elements.

Equivalence classes for compound data types can be derived as a product of those of its components. However, to reduce the number of equivalence classes, one may use equivalence classes derived from its components.

Compound data types: Input data may also be modeled as a compound type. Any input data value that has more than one independent attribute is a compound type. For example, arrays in Java and records, or structures in C++, are compound types. Such input types may arise while testing components of an application such as a function or an object. While generating equivalence classes for such inputs, one must consider legal and illegal values for each component of the structure. The next example illustrates the derivation of equivalence classes for an input variable that has a compound type.

Example 3.6 A student record system, say *S*, maintains and processes data on students in a university. The data is kept in a database, in the form of student records, one record for each student, and is read into appropriate variables prior to processing and rewriting back into the database. Data for newly enrolled students is the input via a combination of mouse and keyboard actions.

One component of *S*, named *transcript*, takes a student record *R* and an integer *N* as inputs and prints *N* copies of the student's transcript corresponding to *R*. For simplicity, let us assume the following structure for *R*.

Program P3.2

```
struct R
  {
  string fName; // First name.
  string lName; //  Last name.
  string cTitle [200]; // Course titles.
  char grades [200]; // Letter grades corresponding
    to course titles.
  }
```

Structure *R* contains a total of four data components. The first two of these components are of a primitive type while the last two being arrays are of compound types. A general procedure for determining the equivalences classes for *transcript* is given in the following section. For now, it should suffice to mention that one step in determining the equivalence classes for *transcript* is to derive equivalence classes for each component of *R*. This can be done by referring to Tables 3.1 and 3.2 and following the guidelines as explained. The classes so derived are then combined using the procedure described in the next section. (See Exercise 3.7.)

Most objects, and applications, require more than one input. In this case, a test input is a collection of values, one for each input. Generating such tests requires partitioning the input domain of the object, or the application, and not simply the set of possible values for one input variable. However, the guidelines in Tables 3.1 and 3.2 assist in partitioning individual variables. The equivalence classes so created are then combined to form a partition of the input domain.

3.3.4 Unidimensional partitioning versus multidimensional partitioning

The input domain of a program can be partitioned in a variety of ways. Here we examine two generic methods for partitioning the input domain and point out their advantages and disadvantages. Examples of both methods are found in the subsequent sections. We are concerned with programs that have two or more input variables.

Equivalence partitioning considering one program variable at a time is referred to as unidimensional partitioning. Multidimensional partitioning refers to a process that accounts for more than one program variable at a time to derive an equivalence partition.

One way to partition the input domain is to consider one input variable at a time. Thus, each input variable leads to a partition of the input domain. We refer to this style of partitioning as *unidimensional* equivalence partitioning or simply *unidimensional* partitioning. In this case, there is some relation R, one for the input domain of each variable. The input domain of the program is partitioned based on R; the number of partitions being equal to the number of input variables and each partition usually contains two or more equivalence classes.

Another way is to consider the input domain $\mathcal{I}$ as the set product of the input variables and define a relation on $\mathcal{I}$. This procedure creates one partition consisting of several equivalence classes. We refer to this method as *multidimensional* equivalence partitioning or simply *multidimensional* partitioning.

Test case selection has traditionally used unidimensional partitioning due to its simplicity and scalability. Multidimensional partitioning leads to a large, sometimes too large, number of equivalence classes that are difficult to manage at least manually. Many of the classes so created might be infeasible in that a test selected from such a class does not satisfy the constraints amongst the input variables. Nevertheless, equivalence classes created using multidimensional partitioning offer an increased variety of tests as is illustrated in the next section. The next example illustrates both unidimensional and multidimensional partitioning.

Example 3.7 Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: $3 \leq x \leq 7$ and $5 \leq y \leq 9$. For uni-dimensional

Figure 3.4 Geometric representation of equivalence classes derived using unidimensional partitioning based on x and y as in (a) and (b), respectively, and using multidimensional partitioning as in (c).

partitioning, we apply the partitioning guidelines in Tables 3.1 and 3.2 to x and y individually. This leads to the following six equivalence classes.

E1:	$x < 3$	E2:	$3 \leq x \leq 7$	E3:	$x > 7$
E4:	$y < 5$	E5:	$5 \leq y \leq 9$	E6:	$y > 9$

Figures 3.4(a) and (b) show the partitions of the input domain based on, respectively, x and y. However, we obtain the following 9 equivalence classes if we consider the input domain to be the product of X and Y, where X and Y denote, respectively, the set of values of variables x and y.

E1:	$x < 3, y < 5$	E2:	$x < 3, 5 \leq y \leq 9$	E3:	$x < 3, y > 9$
E4:	$3 \leq x \leq 7, y < 5$	E5:	$3 \leq x \leq 7, 5 \leq y \leq 9$	E6:	$3 \leq x \leq 7, y > 9$
E7:	$x > 7, y < 5$	E8:	$x > 7, 5 \leq y \leq 9$	E9:	$x > 7, y > 9$

From the test selection perspective, the two unidimensional partitions generate six representative tests, one for each equivalence class. However, we obtain nine representative tests when using multidimensional partitioning. Which set of tests is better in terms of fault detection depends on the type of application.

The tests selected using multidimensional partitioning are likely to test an application more thoroughly than those selected using uni-dimensional partitioning. On the other hand, the number of equivalence classes created using multidimensional partitioning increases exponentially in the number of input variables. (See Execrises 3.8, 3.9, 3.10, and 3.11.)

3.3.5 A systematic procedure

While equivalence partitioning is mostly applied manually, and to derive unit tests, following a systematic procedure will likely aid in the derivation of effective partitions.

Test selection based on equivalence partitioning can be used for small and large applications. For applications or objects involving a few variables, say 3–5, test selection could be done manually. However, as the application or object size increase to, say, 25 or more input variables, manual application becomes increasingly difficult and error prone. In such situations, the use of a tool for test selection is highly recommended.

The following steps are helpful in creating the equivalence classes given program requirements. The second and third steps could be followed manually or automated. The first step below, identification of the input domain, will most likely be executed manually unless the requirements have been expressed in a formal specification language such as Z in which case this step can also be automated.

1. *Identify the input domain*: Read the requirements carefully and identify all input and output variables, their types, and any conditions associated with their use. Environment variables, such as class variables used in the method under test and environment variables in Unix, Windows, and other operating systems, also serve as input variables. Given the set of values each variable can assume, an approximation to the input domain is the product of these sets. As we will see in Step 4, constraints specified in requirements and the design of the application to be tested will likely eliminate several elements from the input domain derived in this step.
2. *Equivalence classing*: Partition the set of values of each variable into disjoint subsets. Each subset is an equivalence class. Together, the equivalence classes based on an input variable partition the input domain. Partitioning the input domain using values of one variable is done based on the expected behavior of the program. Values for which the program is expected to behave in the "same way" are grouped together. Note that "same way" needs to be defined by the tester. Examples given above illustrate such grouping.
3. *Combine equivalence classes*: This step is usually omitted and the equivalence classes defined for each variable are directly used to select test

cases. However, by not combining the equivalence classes, one misses the opportunity to generate useful tests.

The equivalence classes are combined using the multidimensional partitioning approach described earlier. For example, suppose that program P takes two integer valued inputs denoted by X and Y. Also suppose that values of X have been partitioned into sets X_1 and X_2 while the values of Y have been partitioned into sets Y_1, Y_2, and Y_3. Taking the set product of $\{X_1, X_2\}$ and $\{Y_1, Y_2, Y_3\}$, we get the following set E of six equivalence classes for P; each element of E is an equivalence class obtained by combining one equivalence class of X with another of Y.

$$E = \{X_1 \times Y_1, X_1 \times Y_2, X_1 \times Y_3, X_2 \times Y_1, X_2 \times Y_2, X_2 \times Y_3\}$$

Note that this step might lead to an unmanageably large number of equivalence classes and hence is often avoided in practice. Ways to handle such an explosion in the number of equivalence classes are discussed in Sections 3.2, 4.3 (in Chapter 4), and in Chapter 6.

An infeasible equivalence class contains inputs that cannot be applied to the program under test.

4. *Identify infeasible equivalence classes*: An infeasible equivalence class is one that contains a combination of input data that cannot be generated during a test. Such an equivalence class might arise due to several reasons. For example, suppose that an application is tested via its GUI, i.e. data is input using commands available in the GUI. The GUI might disallow invalid inputs by offering a palette of valid inputs only. There might also be constraints in the requirements that render certain equivalence classes infeasible.

   Infeasible data is a combination of input values that cannot be input to the application under test. Again, this might happen due to the filtering of the invalid input combinations by a GUI. While it is possible that some equivalence classes are wholly infeasible, the likely case is that each equivalence class contains a mix of testable and infeasible data.

   In this step, we remove from E equivalence classes the ones that contain infeasible inputs. The resulting set of equivalence classes is then used for the selection of tests. Care needs to be exercised while selecting tests as some data in the reduced E might also be infeasible.

Example 3.8 Let us now apply the equivalence partitioning procedure described earlier to generate equivalence classes for the software control portion of a boiler controller. A simplified set of requirements for the control portion are as follows.

Consider a boiler control system (*BCS*). The control software of *BCS*, abbreviated as *CS*, is required to offer one of several options. One of the options, *C* (for control), is used by a human operator to

give one of three commands (cmd): change the boiler temperature (temp), shut down the boiler (shut), and cancel the *request* (cancel). Command *temp* causes *CS* to ask the operator to enter the amount by which the temperature is to be changed (tempch). Values of *tempch* are in the range [−10, 10] in increments of 5 degrees Fahrenheit. A temperature change of 0 is not an option.

Selection of option *C* forces the *BCS* to examine *V*. If *V* is set to *GUI*, the operator is asked to enter one of the three commands via a GUI. However, if *V* is set to *file*, *BCS* obtains the command from a command file.

The command file may contain any one of the three commands, together with the value of the temperature to be changed if the command is *temp*. The file name is obtained from variable *F*. Values of *V* and *F* can be altered by a different module in *BCS*. In response to *temp* and *shut* commands, the control software is required to generate appropriate signals to be sent to the boiler heating system.

We assume that the control software is to be tested in a simulated environment. The tester takes on the role of an operator and interacts with the CS via a GUI. The GUI forces the tester to select from a limited set of values as specified in the requirements. For example, the only options available for the value of *tempch* are −10, −5, 5, and 10. We refer to these four values of *tempch* as *t_valid* while all other values as *t_invalid*. Figure 3.5 is a schematic of the GUI, the control software under test, and the input variables.

Identify the input domain: The first step in generating equivalence partitions is to identify the (approximate) input domain. Recall that the domain identified in this step will likely be a superset of the true input domain of the control software. First we examine the

While deriving equivalence classes, it is useful to consider illegal values as possible inputs.

Figure 3.5 Inputs to the boiler control software. *V* and *F* are environment variables. Values of *cmd* (command) and *tempch* (temperature change) are input via the GUI or a data file depending on *V*. *F* specifies the data file.

requirements and identify input variables, their types, and values. These are listed in the following table.

Variable	Kind	Type	Value(s)
V	Environment	Enumerated	$\{GUI, file\}$
F	Environment	String	A file name
cmd	Input via GUI or file	Enumerated	$\{temp, cancel, shut\}$
$tempch$	Input via GUI or file	Enumerated	$\{-10, -5, 5, 10\}$

Considering that each variable name defines a set, we derive the following set as a product of the four variables listed above.

$$S = V \times F \times cmd \times tempch.$$

The input domain of BCS, denoted by $\mathcal{I}$, contains S. Sample values in $\mathcal{I}$, and in S, are given below where the underscore character (_) denotes a "don't care" value.

$$(GUI, _, temp, -5), (GUI, _, cancel, _), (file, cmd_file, shut, _)$$

The following 4-tuple belongs to $\mathcal{I}$ but not to S.

$$(file, cmd_file, temp, 0)$$

Equivalence classing: Equivalence classes for each variable are given in the table below. Recall that for variables that are of an enumerated type, each value belongs to a distinct partition.

Variable	Partition
V	$\{\{GUI\}, \{file\}, \{undefined\}\}$
F	$f_valid, f_invalid$
cmd	$\{\{temp\}, \{cancel\}, \{shut\}, c_invalid\}$
$tempch$	$\{\{-10\}, \{-5\}, \{5\}, \{10\}, t_invalid\}$

f_valid denotes a set of names of files that exist, *f_invalid* denotes the set of names of nonexistent files, *c_invalid* denotes the set of invalid commands specified in F, *t_invalid* denotes the set of out of range values of *tempch* in the file, and *undefined* indicates that the environment variable V is undefined. Note that *f_valid*, *f_invalid*, *c_invalid*, and *t_invalid* denote sets of values, whereas *undefined* is a singleton indicating that V is undefined.

Combine equivalence classes: Variables V, F, *cmd*, and *tempch* have been partitioned into 3, 2, 4, and 5 subsets, respectively. Set product of these four variables leads to a total of $3 \times 2 \times 4 \times 5 = 120$ equivalence classes. A sample follows.

$\{(GUI, f_valid, temp, -10)\}$, $\{(GUI, f_valid, temp, t_invalid)\}$,
$\{(file, f_invalid, c_invalid, 5)\}$, $\{(undefined, f_valid, temp, t_invalid)\}$,
$\{(file, f_valid, temp, -10)\}$, $\{(file, f_valid, temp, -5)\}$

Note that each of the classes listed above represents an infinite number of input values for the control software. For example, $\{(GUI, f_valid, temp, -10)\}$ denotes an infinite set of values obtained by replacing *f_valid* by a string that corresponds to the name of an existing file. As we shall see later, each value in an equivalence class is a potential test input for the control software.
Discard infeasible equivalence classes: Note that the amount by which the boiler temperature is to be changed is needed only when the operator selects *temp* for *cmd*. Thus, several equivalence classes that match the following template are infeasible.

$\{(V, F, \{cancel, shut, c_invalid\}, t_valid \cup t_invalid)\}$

This parent–child relationship between *cmd* and *tempch* renders infeasible a total of $3 \times 2 \times 3 \times 5 = 90$ equivalence classes.

Next, we observe that the GUI does not allow invalid values of temperature change to be input. This leads to two more infeasible equivalence classes given below.

$\{(GUI, f_valid, temp, t_invalid)\}$ $\{(GUI, f_invalid, temp, t_invalid)\}$	*The GUI does not allow invalid values of temperature change to be input.*

Continuing with a similar argument, we observe that a carefully designed application might not ask for the values of *cmd* and *tempch* when $V = file$ and F contains a file name that does not exist. In this case, five additional equivalence classes are rendered infeasible. Each of these five classes is described by the following template:

$\{(file, f_invalid, temp, t_valid \cup t_invalid)\}$.

Along similar lines as above, we can argue that the application will not allow values of *cmd* and *tempch* to be input when V is undefined. Thus, yet another five equivalence classes that match the following template are rendered infeasible:

$\{(undefined, _, temp, t_valid \cup t_invalid)\}$.

Note that even when V is undefined, F can be set to a string that may or may not represent a valid file name.

The above discussion leads us to discard a total of $90 + 2 + 5 + 5 = 102$ equivalence classes. We are now left with only 18 testable equivalence classes. Of course, our assumption in discarding 102 classes is that the application is designed so that certain combinations of input values are impossible to achieve while testing the control software. In the absence of this assumption, all 120 equivalence classes are potentially testable.

The set of 18 testable equivalence classes match the seven templates listed below. The "_" symbol indicates, as before, that the data can be input but is not used by the control software, and *NA* indicates that the data cannot be input to the control software because the GUI does not ask for it.

$\{(GUI, f_valid, temp, t_valid)\}$	*4 equivalence classes.*
$\{(GUI, f_invalid, temp, t_valid)\}$	*4 equivalence classes.*
$\{(GUI, _, cancel, NA)\}$	*2 equivalence classes.*
$\{(file, f_valid, temp, \text{t_valid} \cup t_invalid)\}$	*5 equivalence classes.*
$\{(file, f_valid, shut, NA)\}$	*1 equivalence classes.*
$\{(file, f_invalid, NA, NA)\}$	*1 equivalence class.*
$\{(undefined, NA, NA, NA)\}$	*1 equivalence class.*

There are several input data tuples that contain don't care values. For example, if $V = GUI$, then the value of F is not expected to have any impact on the behavior of the control software. However, as explained in the following section, a tester must interpret such requirements carefully.

3.3.6 Test selection

An equivalence class might contain several infeasible tests. This could happen, for example, due to the presence of a GUI that correctly prevents certain values to be input.

Given a set of equivalence classes that form a partition of the input domain, it is relatively straightforward to select tests. However, complications could arise in the presence of infeasible data and don't care values. In the most general case, a tester simply selects one test that serves as a representative of each equivalence class. Thus, for example, we select four tests, one from each equivalence class defined by the *pCat* relation in Example 3.4. Each test is a string that denotes the make and model of a printer belonging to one of the three classes or is an invalid string. Thus, the following is a set of four tests selected from the input domain of program *pTest*.

$$T = \{\text{HP cp1700}, \text{Canon Laser Shot LBP 3200}, \\ \text{Epson Stylus Photo RX600}, \text{My Printer} \\ \}$$

While testing `pTest` against tests in T, we assume that if `pTest` correctly selects a script for each printer in T, then it will select correct scripts for all printers in its database. Similarly, from the six equivalence classes in Example 3.5, we generate the following set T consisting of six tests each of the form (w, f), where w denotes the value of an input word and f a file name.

$$T = \{(\text{Love}, \text{my-dict}), (\text{Hello}, \text{does-not-exist}), (\text{Bye}, \text{empty-file}), \\ (\epsilon, \text{my-dict}), (\epsilon, \text{does-not-exist}), (\epsilon, \text{empty-file}) \\ \}$$

In the test above, ϵ denotes an empty, or a null, string, implying that the input word is a null string. Values of f *my-dict*, *does-not-exist*, and *empty-file* correspond to names of files that, respectively, exist, do not exist, and exist but is empty.

Selection of tests for the boiler control software in Example 3.8 is a bit more tricky due to the presence of infeasible data and don't care values. The next example illustrates how to proceed with test selection for the boiler control software.

Example 3.9 Recall that we began by partitioning the input domain of the boiler control software into 120 equivalence classes. A straightforward way to select tests would have been to pick one representative of each class. However, due to the existence of infeasible equivalence classes, this process would lead to tests that are infeasible. We therefore derive tests from the reduced set of equivalence classes. This set contains only feasible classes. Table 3.3 lists 18 tests derived from the 18 equivalence classes listed earlier in Example 3.8. The equivalence classes are labeled as E1, E2, and so on for ease of reference.

While deriving the test data in Table 3.3, we have specified arbitrary values of variables that are not expected to be used by the control software. For example, in E9, the value of F is not used. However, to generate a complete test data item, we have arbitrarily assigned to F a string that represents the name of an existing file. Similarly, the value of *tempch* is arbitrarily set to -5.

Table 3.3 Test data for the control software of a boiler control system in Example 3.8.

ID	Equivalence class* $\{(V, F, cmd, tempch)\}$	Test data† $(V, F, cmd, tempch)$
E1	$\{(GUI, f_valid, temp, t_valid)\}$	$(GUI, a_file, temp, -10)$
E2	$\{(GUI, f_valid, temp, t_valid)\}$	$(GUI, a_file, temp, -5)$
E3	$\{(GUI, f_valid, temp, t_valid)\}$	$(GUI, a_file, temp, 5)$
E4	$\{(GUI, f_valid, temp, t_valid)\}$	$(GUI, a_file, temp, 10)$
E5	$\{(GUI, f_invalid\ temp, t_valid)\}$	$(GUI, no_file, temp, -10)$
E6	$\{(GUI, f_invalid\ temp, t_valid)\}$	$(GUI, no_file, temp, -10)$
E7	$\{(GUI, f_invalid\ temp, t_valid)\}$	$(GUI, no_file, temp, -10)$
E8	$\{(GUI, f_invalid\ temp, t_valid)\}$	$(GUI, no_file, temp, -10)$
E9	$\{(GUI, __, cancel, \text{NA})\}$	$(GUI, a_file, cancel, -5)$
E10	$\{(GUI, __, cancel, \text{NA})\}$	$(GUI, no_file, cancel, -5)$
E11	$\{(file, f_valid, temp, t_valid)\}$	$(file, a_file, temp, -10)$
E12	$\{(file, f_valid, temp, t_valid)\}$	$(file, a_file, temp, -5)$
E13	$\{(file, f_valid, temp, t_valid)\}$	$(file, a_file, temp, 5)$
E14	$\{(file, f_valid, temp, t_valid)\}$	$(file, a_file, temp, 10)$
E15	$\{(file, f_valid, temp, t_invalid)\}$	$(file, a_file, temp, -25)$
E16	$\{(file, f_valid, temp, \text{NA})\}$	$(file, a_file, shut, 10)$
E17	$\{(file, f_invalid, \text{NA}, \text{NA})\}$	$(file, no_file, shut, 10)$
E18	$\{(undefined, __, \text{NA}, \text{NA})\}$	$(undefined, no_file, shut, 10)$

*__: Don't care. NA : Input not allowed.
†*a_file*: File exists. *no_file*: File does not exist.

Elements of an equivalence class might contain don't care values. These "values" may need to be converted to actual values when testing the program.

The don't care values in an equivalence class must be treated carefully. It is from the requirements that we conclude that a variable is or is not don't care. However, an incorrect implementation might actually make use of the value of a don't care variable. In fact, even a correct implementation might use the value of a variable marked as don't care. This latter case could arise if the requirements are incorrect or ambiguous. In any case, it is recommended that one should assign data to don't care variables, given the possibility that the program may be erroneous and hence make use of the assigned data value.

In Step 3 of the procedure, to generate equivalence classes of an input domain, we suggested the product of the equivalence classes derived for each input variable. This procedure may lead to a large number of equivalence classes. One approach to test selection that avoids the explosion in the number of equivalence classes is to *cover* each equivalence class of each variable by a small number of tests. We say that a test input *covers*

an equivalence class E for some variable V, if it contains a representative of E. Thus, one test input may cover multiple equivalence classes one for each input variable. The next example illustrates this method for the boiler example.

Example 3.10 In Example 3.8, 3, 2, 4, and 5 equivalence classes were derived, respectively, for variables V, F, *cmd*, and *tempch*. The total number of equivalence classes is only 14; compare this with 120 derived using the product of individual equivalence classes. Note that we have five equivalence classes, and not just two, for variable *tempch*, because it is an enumerated type. The following set T of five tests cover each of the 14 equivalence classes.

$$\begin{aligned} T = \{ & (GUI, a_file, temp, -10), (GUI, no_file, temp, -5), \\ & (file, a_file, temp, 5), (file, a_file, cancel, 10), \\ & (undefined, a_file, shut, -10) \\ & \} \end{aligned}$$

You may verify that indeed the tests listed above cover each equivalence class of each variable. While small, the above test set has several weaknesses. While the test covers all individual equivalence classes, it fails to consider the semantic relations amongst the different variables. For example, the last of the tests listed above will not be able to test the *shut* command assuming that the value *undefined* of the environment variable V is processed correctly.

Several other weaknesses can be identified in T of Example 3.10. The lesson to be learned from this example is that one must be careful, i.e. consider the relationships amongst variables, while deriving tests that cover equivalence classes for each variable. In fact, it is easy to show that a small subset of tests from Table 3.3 will cover all equivalence classes of each variable and satisfy the desired relationships amongst the variables. (See Exercise 3.13.)

3.3.7 Impact of GUI design

A GUI might prevent the input of illegal values of some variables. Nevertheless, the GUI itself must be tested to ensure that this is indeed true.

Test selection is usually affected by the overall design of the application under test. Prior to the development of graphical user interfaces (GUI), data was input to most programs in textual forms through typing on a keyboard, or many many years ago, via punched cards. Most applications of today, either new or refurbished, exploit the advancements in GUI to make interaction with programs much easier and safer than before. Test

design must account for the constraints imposed on data by the front-end application GUI.

While designing equivalence classes for programs that obtain input exclusively from a keyboard, one must account for the possibility of errors in data entry. For example, suppose that the requirement for an application A places a constraint on an input variable X such that it can assume integral values in the range [0, 0.4]. However, a user of this application may inadvertently enter a value for X that is out of range. While the application is supposed to check for incorrect inputs, it might fail to do so. Hence, in test selection using equivalence partitioning, one needs to test the application using at least three values of X, one that falls in the required range and two that fall outside the range on either side. Thus, three equivalence classes for X are needed. Figure 3.6(a) illustrates this situation where the "Incorrect values" portion of the input domain contains the out of range values for X.

However, suppose that all data entry to application A is via a GUI front end. Suppose also that the GUI offers exactly five correct choices to the user for X. In such a situation, it is impossible to test A with a value of X that is out of range. Hence only the correct values of X will be input to A. This situation is illustrated in Figure 3.6(b).

A GUI serving as an interface to a program could reduce its input domain.

Of course, one could dissect the GUI from the core application and test the core separately with correct and incorrect values of X. But any error so discovered in processing incorrect values of X might be meaningless because, as a developer may argue, in practice the GUI would prevent the

Figure 3.6 Restriction of the input domain through careful design of the GUI. Partitioning of the input domain into equivalence classes must account for the presence of GUI as shown in (b) and (c). GUI-A protects all variables against incorrect input while GUI-B does allow the possibility of incorrect input for some variables.

core of A from receiving an invalid input for X. In such a situation, there is no need to define an equivalence class that contains incorrect values of an input variable.

In some cases, a GUI might ask the user to type in the value of a variable in a text box. In a situation like this one must certainly test the application with one or more incorrect values of the input variable. For example, while testing, it is recommended that one use at least three values for X. In case the behavior of A is expected to be different for each integer within the range [0, 4], then A should be tested separately for each value in the range as well as at least two values outside the range. However, one need not include incorrect values of variables in a test case if the variable is protected by the GUI against incorrect input. This situation is illustrated in Figure 3.6(c) where the subset labeled 1 in the set of incorrect values need not be included in the test data while values from subset labeled 2 must be included.

The above discussion leads to the conclusion that test design must take into account GUI design. In some cases, GUI design requirements could be dictated by test design. For example, GUI design might require that whenever possible only valid values be offered as options to the user. Certainly, the GUI needs to be tested separately against such requirements. The boiler control example given earlier shows that the number of tests can be reduced significantly if the GUI prevents invalid inputs from getting into the application under test.

Note that the tests derived in the boiler example make the assumption that the GUI has been correctly implemented and does prohibit incorrect values from entering the control software.

3.4 Boundary Value Analysis

Experience indicates that programmers make mistakes in processing values at or near the boundaries of equivalence classes. For example, suppose that method M is required to compute a function f_1 when the condition $x \leq 0$ is satisfied by input x and function f_2 otherwise. However, M has an error due to which it computes f_1 for $x < 0$ and f_2 otherwise. Obviously, this fault is revealed, though not necessarily, when M is tested against $x = 0$ but not if the input test set is, for example, $\{-4, 7\}$ derived using equivalence partitioning. In this example, the value $x = 0$, lies at the boundary of the equivalence classes $x \leq 0$ and $x > 0$.

Programmers often make errors in handling values of variables at or near the boundaries. Hence, it is important to derive tests that lie at or near the boundaries of program variables.

Boundary value analysis is a test selection technique that targets faults in applications at the boundaries of equivalence classes. While equivalence partitioning selects tests from within equivalence classes, boundary

value analysis focuses on tests at and near the boundaries of equivalence classes. Certainly, tests derived using either of the two techniques may overlap.

While tests derived using boundary value analysis will likely be in one or more equivalence partitions, they might not have been selected using the equivalence partitioning method.

Test selection using boundary value analysis is generally done in conjunction with equivalence partitioning. However, in addition to identifying boundaries using equivalence classes, it is also possible and recommended that boundaries be identified based on the relations amongst the input variables. Once the input domain has been identified, test selection using boundary value analysis proceeds as follows.

1. Partition the input domain using unidimensional partitioning. This leads to as many partitions as there are input variables. Alternately, a single partition of an input domain can be created using multidimensional partitioning.
2. Identify the boundaries for each partition. Boundaries may also be identified using special relationships amongst the inputs.
3. Select test data such that each boundary value occurs in at least one test input. An illustrative example to generate tests using boundary value analysis follows.

Example 3.11 This simple example is to illustrate the notion of boundaries and selection of tests at and near the boundaries. Consider a method *fP*, brief for *find Price*, that takes two inputs *code* and *qty*. The item code is represented by the integer *code* and the quantity purchased by another integer variable *qty*. *fP* accesses a database to find and display the unit price, description, and the total price of the item corresponding to *code*. *fP* is required to display an error message, and return, if either of the two inputs is incorrect. We are required to generate test data to test *fP*.

We start by creating equivalence classes for the two input variables. Assuming that an item code must be in the range [99, 999] and quantity in the range [1, 100], we obtain the following equivalence classes.

Equivalence classes for *code*:	E1: Values less than 99.
	E2: Values in the range.
	E3: Values greater than 999.
Equivalence classes for *qty*:	E4: Values less than 1.
	E5: Values in the range.
	E6: Values greater than 100.

Figure 3.7 shows the equivalence classes for *code* and *qty* and the respective boundaries. Notice that there are two boundaries each

Figure 3.7 Equivalence classes and boundaries for variables (a) *code* and (b) *qty* in Example 3.11. Values at and near the boundary are listed and marked with an "x" and "*", respectively.

In some cases each equivalence class might contain exactly one element which is also at the boundary of the variable under consideration. The equivalence class containing a sole empty string is one example of such a class.

for *code* and *qty*. Marked in the figure are six points for each variable, two at the boundary marked "x", and four near the boundaries marked "*". Test selection based on the boundary value analysis technique requires that tests must include, for each variable, values at and around the boundary. Usually, several sets of tests will satisfy this requirement. Consider the following set.

$$
\begin{aligned}
T = \{ & t_1 : (code = 98, qty = 0), \\
& t_2 : (code = 99, qty = 1), \\
& t_3 : (code = 100, qty = 2), \\
& t_4 : (code = 998, qty = 99), \\
& t_5 : (code = 999, qty = 100), \\
& t_6 : (code = 1000, qty = 101) \\
\}
\end{aligned}
$$

Each of the six values for each variable is included in the tests above. Notice that the illegal values of *code* and *qty* are included in the same test. For example, tests t_1 and t_6 contain illegal values for both *code* and *qty* and require that *fP* display an error message when executed against these tests. Tests t_2, t_3, t_4, and t_5 contain legal values.

While T is a minimal set of tests that include all boundary and near-boundary points for *code* and *qty*, it is not the best when evaluated against its ability to detect faults. Consider, for example, the following faulty code skeleton for method *fP*.

```
public void fP(int code, qty)
{
  if (code<99 || code>999)
  {display_error(``Invalid code''); return;}
  // Validity check for qty is missing!
  // Begin processing code and qty.
  ⋮
}
```

When *fP* that includes the above code segment is executed against t_1 or t_6, it will correctly display an error message. This behavior would indicate that the value of *code* is incorrect. However, these two tests fail to check that the validity check on *qty* is missing from the program. None of the other tests will be able to reveal the missing code error. By separating the correct and incorrect values of different input variables, we increase the possibility of detecting the missing code error.

One could improve the chances of detecting a missing code by separating the tests for legal and illegal values of different input variables.

It is also possible that the check for the validity of *code* is missing but that for checking the validity of *qty* exists and is correct. Keeping the two possibilities in view, we generate the following four tests that can safely replace t_1 and t_6 listed above. The following two tests are also derived using boundary value analysis.

$t_7 = (code = 98, qty = 45)$
$t_8 = (code = 1000, qty = 45)$
$t_9 = (code = 250, qty = 0)$
$t_{10} = (code = 250, qty = 101)$

A test suite for *fP*, derived using boundary value analysis, now consists of $t_2, t_3, t_4, t_5, t_7, t_8, t_9$, and t_{10}. If there is a reason to believe that tests that include boundary values of both inputs might miss some error, then tests t_2 and t_6 must also be replaced to avoid the boundary values of *code* and *qty* from appearing in the same test. (See Exercise 3.17.)

Example 3.12 Consider a method named *textSearch* to search for a non-empty string *s* in text *txt*. Position of characters in *txt* begins with 0 representing the first character, 1 the next character, and so on. Both *s* and *txt* are supplied as parameters to *textSearch*. The method returns an integer *p* such that if $p \geq 0$, then it denotes the starting position of *s* in *txt*. A negative value of *p* implies that *s* is not found in *txt*.

To apply the boundary value analysis technique, we first construct the equivalence classes for each input variable. In this case, both inputs are strings and no constraints have been specified on their length or contents. Based on the guidelines in Tables 3.1 and 3.2, we arrive at the following four equivalence classes for *s* and *txt*.

It helps to construct equivalence classes for each input variables. These classes can then be used to identify values at and near the boundaries of each variable.

Equivalence classes for *s*: E_1: empty string, E_2: non-empty string.
Equivalence classes for *txt*: E_3: empty string, E_4: non-empty string.

It is possible to define boundaries for a string variable based on its length and semantics. In this example, the lower bound on the lengths of both *s* and *txt* is 0 and hence this defines one boundary for each input. No upper bound is specified on lengths of either variable. Hence, we obtain only one boundary case for each of the two inputs. However, as is easy to observe, E_1 and E_3 contain exactly the boundary case. Thus, tests derived using equivalence partitioning on input variables are the same as those derived by applying boundary value analysis.

Let us now partition the output space of *textSearch* into equivalence classes. We obtain the following two equivalence classes for *p*:

E_5: $p < 0$, E_6: $p \geq 0$.

To obtain an output that belongs to E_5, the input string *s* must not be in *txt*. Also, to obtain an output value that falls into E_6, input *s* must be in *txt*. As both E_5 and E_6 are open ranges, we obtain only one boundary, $p = 0$ based on E_5 and E_6. A test input that causes a correct *textSearch* to output $p = 0$ must satisfy two conditions: (i) *s* must be in *txt* and (ii) *s* must appear at the start of *txt*. These two conditions allow us to generate the following test input based on boundary value analysis:

$s =$ "Laughter" and $txt =$ "Laughter is good for the heart."

Based on the six equivalence classes and two boundaries derived, we obtain the following set *T* of three test inputs for *textSearch*.

$$
\begin{aligned}
T = \{\ & t_1 : (s = \epsilon, txt = \text{"Laughter is good for the heart."}),\\
& t_2 : (s = \text{"Laughter"}, txt = \epsilon),\\
& t_3 : (s = \text{"good for"}, txt = \text{"Laughter is good for the heart."}),\\
& t_4 : (s = \text{"Laughter"}, txt = \text{"Laughter is good for the heart."})\}
\end{aligned}
$$

It is easy to verify that tests t_1 and t_2 cover equivalence classes E_1, E_2, E_3, E_4, and E_5 and t_3 covers E_6. Test t_4 is necessitated by the conditions imposed by boundary value analysis. Notice that none of the six equivalence classes requires us to generate t_4. However, boundary value analysis, based on output equivalence classes E_5 and E_6, requires us to generate a test in which s occurs at the start of txt.

Test t_4 suggests that we add to T yet another test t_5 where s occurs at the end of txt.

$t_5 : (s =$ "heart.", $txt =$ "Laughter is good for the heart.")

None of the six equivalence classes derived above, or the boundaries of those equivalence classes, directly suggest t_5. However, both t_4 and t_5 aim at ensuring that *textSearch* behaves correctly when s occurs at the boundaries of txt.

Having tests on the two boundaries determined using s and txt, we now examine test inputs that are near the boundaries. Four such cases are listed below.

- s starts at position 1 in txt. Expected output: $p = 1$.
- s ends at one character before the last character in txt. Expected output: $p = k$, where k is the position from where s starts in txt.
- All but the first character of s occur in txt starting at position 0. Expected output: $p = -1$.
- All but the last character of s occur in txt at the end. Expected output: $p = -1$.

The following tests, added to T, satisfy the four conditions listed above.

$t_6 : (s =$ "aughter", $txt =$ "Laughter is good for the heart."),

$t_7 : (s =$ "heart", $txt =$ "Laughter is good for the heart."),

$t_8 : (s =$ "gLaughter", $txt =$ "Laughter is good for the heart."),

$t_9 : (s =$ "heart.d", $txt =$ "Laughter is good for the heart.").

We now have a total of nine tests of which six are derived using boundary value analysis. Points on and off the boundary are shown in Figure 3.8. Two boundaries are labeled 1 and 2 and four points off the boundaries are labeled 3, 4, 5, and 6.

Figure 3.8 c_0 through c_7 are the eight characters in *txt* starting at the leftmost position 0. s_0s_1 is the input string *s*. Shown here is the positioning of *s* with respect to *txt* at the two boundaries, labeled 1 and 2, and at four points near the boundary, labeled 3 through 6.

Next, one might try to obtain additional tests by combining the equivalence classes for *s* and *txt*. This operation leads to four equivalence classes for the input domain of *textSearch* as listed below.

$E_1 \times E_3, E_1 \times E_4, E_2 \times E_3, E_2 \times E_4.$

Tests t_1, t_2 and t_3 cover all the four equivalence classes except for $E_1 \times E_3$. We need the following test to cover $E_1 \times E_3$.

$t_6 : (s = \epsilon, txt = \epsilon).$

Thus, combining the equivalence classes of individual variables has led to the selection of an additional test case. Of course, one could have derived t_6 based on E_1 and E_3 also. However, coverage of $E_1 \times E_3$ requires t_6, whereas coverage of E_1 and E_3 separately does not. Notice also that test t_4 has a special property that is not required to hold if we were to generate a test that covers the four equivalence classes obtained by combining the equivalence classes for individual variables.

Relationships among input variables ought to be examined to obtain boundaries that might otherwise be not obvious.

The previous example leads to the following observations:

- Relationships amongst the input variables must be examined carefully while identifying boundaries along the input domain. This examination may lead to boundaries that are not evident from equivalence classes obtained from the input and output variables.
- Additional tests may be obtained when using a partition of the input domain obtained by taking the product of equivalence classes created using individual variables.

3.5 Category-Partition Method

The category-partition method, and an associated tool, help ease the implementation of equivalence partitioning and boundary value analysis.

The category-partition method is a systematic approach to the generation of tests from requirements. The method consists of a mix of manual and automated steps. Here, we describe the various steps in using the category-partition method and illustrate with a simple running example.

3.5.1 Steps in the category-partition method

The category-partition method consists of eight steps as shown in Figure 3.9. In essence, a tester transforms requirements into test specifications. These test specifications consist of categories corresponding to program inputs and environment objects.

Each category is partitioned into choices that correspond to one or more values for the input or the state of an environment object. Test specifications also contain constraints on the choices so that only reasonable and valid sets of tests are generated. Test specifications are input to a test frame generator that produces a number of test frames from which test scripts are generated. A test frame is a collection of choices, one corresponding to each category. A test frame serves as a template for one or more test cases that are combined into one or more test scripts.

Test specifications are used to generate test frames. Test scripts are generated from test frames. A test script may at least partially automate the task of applying a test input to the program under test.

Figure 3.9 Steps in the generation of tests using the category-partition method. Tasks in solid boxes are performed manually and generally difficult to automate. Dashed boxes indicate tasks that can be automated.

A running example is used here to describe the steps shown in Figure 3.9. We use the *findPrice* function used in Example 3.11. The requirements for the *findPrice* function are expanded below for the purpose of illustrating the category-partition method.

Function: findPrice

Syntax: fP(code, quantity, weight)

Description:

findPrice takes three inputs: *code*, *qty*, and *weight*. Item code is represented by a string of eight digits contained in variable *code*. The quantity purchased is contained in *qty*. The weight of the item purchased is contained in *weight*.

Function *fP* accesses a database to find and display the unit price, the description, and the total price of the item corresponding to *code*. *fP* is required to display an error message, and return, if either of the three inputs is incorrect. As indicated below, the leftmost digit of *code* decides how the values of *qty* and *weight* are to be used. *code* is an 8-digit string that denotes product type. *fP* is concerned with only the leftmost digit that is interpreted as follows.

Leftmost digit	Interpretation
0	Ordinary grocery items such as bread, magazines, and soap.
2	Variable-weight items such as meats, fruits, and vegetables.
3	Health-related items such as Tylenol, Bandaids, and Tampons.
5	Coupon; digit 2 (dollars), 3 and 4 (cents) specify the discount.
1, 6–9	Unused

The use of parameters *qty* and *weight* depends on the leftmost digit in *code*. *qty* indicates the quantity purchased, an integer, when the leftmost digit is 0 or 3; *weight* is ignored. *weight* is the weight of the item purchased when the leftmost digit is 2; *qty* is ignored. *qty* is the value of the discount when the leftmost digit is 5; again *weight* is ignored. We assume that digits 1, and 6 through 9, are ignored. When the leftmost digit is 5, the second digit from the left specifies the dollar amount and the third and fourth digits the cents.

Step 1: Analyze specification

Prior to generating test scripts, one needs to identify independently testable units of a program.

In this step, the tester identifies each functional unit that can be tested separately. For large systems, a functional unit may correspond to a subsystem that can be tested independently. The subsystem can be further subdivided leading eventually to independently testable subunits. The subdivision process terminates depending on what is to be tested.

Example 3.13 In this example, we assume that *fP* is an independently testable subunit of an application. Thus, we will derive tests for *fP*.

Step 2: Identify categories

For each testable unit, the given specification is analyzed and the inputs isolated. In addition, objects in the environment, e.g. a file, also need to be identified.

A characteristic of an input or an environment variable is considered a category.

Next, we determine characteristics of each parameter and environmental object. A characteristic is also referred to as a *category*. While some characteristics are stated explicitly, others might need to be derived by a careful examination of the specification.

Example 3.14 We note that *fP* has three input parameters: *code*, *qty*, and *weight*. The specification mentions various characteristics of these parameters such as their type and interpretation. Notice that *qty* and *weight* are related to *code*. The specification does mention the types of different parameters but does not indicate bounds on *qty* and *weight*.

The database accessed by *fP* is an environment object. No characteristics of this object are mentioned in the specification. However, for a thorough testing of *fP*, we need to make assumptions about the existence or non-existence of items in the database. We identify the following categories for *fP*.

code: length, leftmost digit, remaining digits
qty: integer-quantity
weight: float-quantity
database: contents

Notice that we have only one category each for *qty*, *weight*, and *database*. In the next step, we see how to partition these categories.

Step 3: Partition categories

For each category the tester determines different cases against which the functional unit must be tested. Each case is also referred to as a *choice*.

One or more cases are expected for each category. It is useful to partition each category into at least two subsets, a set containing correct values and another consisting of erroneous values.

Each category may generate one or more cases against which a functional unit is to be tested.

In the case of a networked application, cases such as network failure ought to be considered. Other possible cases of failure, e.g. database unavailable, also need to be considered. It is for the tester to think of reasonable situations, both valid and invalid, that might arise when the unit under test is in actual use.

Example 3.15 Following is a summary of the various inputs, environment objects, categories, and partitions.

Parameters:
code:
- length
  - valid (8-digits)
  - invalid (less than or greater than 8-digits)
- leftmost digit
  - 0
  - 2
  - 3
  - 5
  - others
- remaining digits
  - valid string
  - invalid string (e.g., 0X5987Y)

qty:
- integer-quantity
  - valid quantity
  - invalid quantity (e.g., 0)

weight:
- float-quantity
  - valid weight
  - invalid weight (e.g., 0)

Environments:
database:
- contents
  - item exists
  - item does not exist

Notice that if *fP* were to operate in a networked environment, then several other cases are possible. We leave these to Exercise 3.20.

Step 4: Identify constraints

There might exist several combinations of choices derived earlier. One needs to specify constraints among choices so that only valid combinations lead to test frames.

A test for a functional unit consists of a combination of choices for each parameter and environment object. Certain combinations might not be possible while others must satisfy specific relationships. In any case, constraints amongst choices are specified in this step. These constraints are used by the test generator to produce only the valid test frames in Step 6.

A constraint is specified using a property list and a selector expression. A property list has the following form:

[property P1, P2,..]

where "property" is a key word and P1, P2, etc. are names of individual properties. Each choice can be assigned a property. A selector expression is a conjunction of pre-defined properties specified in some property list. A selector expression takes one of the following forms:

[`if` P1]
[`if` P1 and P2 and . . .]

The above two forms can be suffixed to any choice. A special property written as [`error`] can be assigned to choices that represent error conditions. Another special property written as [single] allows the tester to specify that the associated choice is not to be combined with choices of other parameters, or environment objects, while generating test frames in Step 6.

Example 3.16 Properties and selector expressions assigned to a few choices sampled from Example 3.15 follow; comment lines start with #.

# Leftmost digit of code	
0	[property ordinary-grocery]
2	[property variable-weight]
# Remaining digits of code	
valid string	[single]
# Valid value of *qty*	
valid quantity	[`if` ordinary-grocery]
# Invalid value of *qty*	
invalid quantity	[error]

Step 5: (Re)write test specification

Having assigned properties and selector expressions to various choices, the tester now writes a complete test specification. The specification is written in a test specification language (TSL) conforming to a precise syntax.

This step may need to be executed more than once in the event the tester is not satisfied with the test frames generated in Step 6. This happens when the tester evaluates the test frames in Step 7 and finds that some test frames are redundant. Such frames may have been generated by combining choices that are either impractical or unlikely to arise in practice. In such an event, the tester rewrites the specification and inputs them for processing in Step 6.

A complete test specification is written in Test Specification Language. Test scripts are eventually derived from this specification.

Example 3.17 A complete test specification for *fP* follows; we have ignored the processing coupon category (see Exercise 3.21). We have used a slight variation of the TSL syntax as given by Ostrand and Balcer (also see Bibliographic notes).

```
Parameters:
code
  length
    valid
    invalid          [error]

Leftmost digit

  0                  [property Ordinary-grocery]
  2                  [property Variable-weight]
  3                  [property Health-related]
  5                  [property Coupon]
Remaining digits
  valid string       [single]
  invalid string     [error]
 qty
    valid quantity       [if ordinary-grocery]
    invalid quantity     [error]
 weight
    valid weight         [if variable-weight]
    invalid weight       [error]
 Environments:
 database
    item exists
    item does not exist  [error]
```

Step 6: Process specification

The TSL specification written in Step 5 is processed by an automatic test frame generator. This results in a number of test frames. The test frames are

analyzed by the tester for redundancy, i.e. they might test the application in the same way. In such cases, one either rewrites the test specifications or simply ignores the redundant frames.

Example 3.18 A sample test frame generated from the test specification in Example 3.17 follows.

Test case 2: (Key = 1.2.1.0.1.1)

Length:	valid
Leftmost digit:	2
Remaining digits:	valid string
qty:	ignored
weight:	3.19
database:	item exists

A test number identifies the test. The key in a test frame indicates the choices used, a 0 indicating no choice. *qty* is ignored as the leftmost digit corresponds to a variable weight item that is priced by weight and not by quantity. (Note that the terminology used in this example differs from that in the original TSL.)

A test frame is not a test case. However, a test case, consisting of specific input values and expected outputs, can be derived from a test frame. It is important to note that the test frame contains information about the environment objects. Hence, it is useful in setting up the environment prior to a test.

Test frames are generated from valid combinations of choices. A test frame may then be transformed in to a script or a set of test cases.

Test frames are generated by generating all possible combinations of choices while satisfying the constraints. Choices that are marked error or single are not combined with others and produce only one test case. It is easy to observe that without any constraints (selector expressions), a total of 128 test frames will be generated from the specification in Example 3.17.

Step 7: Evaluate generator output

In this step, the tester examines the test frames for any redundancy or missing cases. This might lead to a modification in the test specification (Step 5) and a return to Step 6.

Step 8: Generate test scripts

Test cases generated from test frames are combined into test scripts. A test script is a grouping of test cases. Generally, test cases that do not require

any changes in settings of the environment objects are grouped together. This enables a test driver to efficiently execute the tests.

That completes our description of the category-partition method. As you may have noticed, the method is essentially a systematization of the equivalence partitioning and boundary value techniques discussed earlier.

Writing a test specification allows a test team to take a close look at the program specifications. For large systems, this task can be divided amongst members of the test team. While a significant part of the method requires manual work, availability of a tool that processes TSL specifications helps in test generation and documentation. It also reduces the chances of errors in test cases.

SUMMARY

In this chapter, we have introduced a class of foundational techniques for generating tests. These techniques have existed for several decades and are used extensively in practice, either knowingly or unknowingly, by testers. The techniques using equivalence partitioning, boundary value analysis, domain testing and cause-effect graphing fall under the category best known as *partition testing*. This form of testing is perhaps the most commonly used strategy for the generation of tests especially during unit testing. These techniques aim to partition the input domain of the program under test into smaller sub-domains. A few tests are then selected from each subdomain.

Exercises

3.1 Let N be the size of a sequence s of integers. Assume that an element of s can be any one of v distinct values. Show that the number of possible sequences is $\Sigma_{i=0}^{N} v^i$.

3.2 An equivalence relation R on set S is reflexive, symmetric, and transitive. Also, R partitions S into equivalence classes. Show that each of the relations defined in Exercises 3.3 and 3.4 is an equivalence relation.

3.3 Derive equivalence classes for the input variables listed below.

a. `int` *pen_inventory*; Current inventory level of writing pens.
b. `string` *planet_name*; Planet name.
c. *operating_system*={"OS X", "Windows XP", "Windows 2000", "Unix", "Linux", "Xinu", "VxWorks"}; Name of an operating system.
d. `printer_class=set` printer_name; `printer_class` p; Set of printer names.
e. `int` *name* [1.10]; An array of at most 10 integers.

3.4 In Example 3.4, suppose now that we add another category of printers, say, "Home and home office (hb)." Define a suitable relation *hb* that partitions the input domain of `pTest` into two equivalence classes. Discuss the overlap of the equivalence classes induced by *hb* with the remaining eight classes defined by the four relations in Example 3.4.

3.5 Consider the following relation

$cl : \mathcal{I} \rightarrow \{yes, no\}$

cl maps an input domain $\mathcal{I}$ of *pTest* in Example 3.4 to the set $\{yes, no\}$. A printer make and model is mapped to *yes* if it is a color laserjet, else it is mapped to *no*. Is *cl* an equivalence relation?

3.6 (a) Why consider classes E2–E6 in Example 3.5 when the correctness of the program corresponding to tests in these classes can be verified by simple inspection? Offer at least two reasons. (b) Are there any additional equivalence classes that one ought to consider while partitioning the input domain of `wordCount`?

3.7 Partition the input domain of the *transcript* component described in Example 3.6 into equivalence classes using the guidelines in Tables 3.1 and 3.2. Note that *transcript* takes two inputs, a record R and an integer N.

3.8 (a) Generate two sets of tests T_1 and T_2 from the partitions created in Example 3.7 using, respectively, uni-dimensional and multidimensional partitioning. Which of the following relations holds amongst T_1 and T_2 that you have created: $T_1 = T_2$, $T_1 \subset T_2$, $T_1 \subseteq T_2$, $T_1 \supset T_2$, $T_1 \supseteq T_2$, and $T_1 \neq T2$? (b) Which of the six relations mentioned could hold between T_1 and T_2 assuming that T_1 is derived from equivalence classes constructed using uni-dimensional partitioning and T_2 using multidimensional partitioning?

3.9 Consider an application *App* that takes two inputs *name* and *age* where *name* is a non-empty string containing at most 20 alphabetic characters and *age* is an integer that must satisfy the constraint $0 \leq age \leq 120$. *App* is required to display an error message if the input value provided for *age* is out of range. It truncates any name that is more than 20 characters in length and generates an error message if an empty string is supplied for *name*.

Partition the input domain using (a) uni-dimensional partitioning and (b) multidimensional partitioning. Construct two sets of test data for *App* using the equivalence classes derived in (a) and in (b).

3.10 Suppose that an application has m input variables and that each variable partitions the input space into n equivalence classes. The multidimensional partitioning approach will divide the input domain into how many equivalence classes?

3.11 An application takes two inputs x and y where $x \leq y$ and $-5 \leq y \leq 4$. (a) Partition the input domain using uni-dimensional and multidimensional partitioning. (b) Derive test sets based on the partitions created in (a).

3.12 In Example 3.8, we started out by calculating the number of equivalence classes to be 120. We did so because we did not account for the parent–child relationship between *cmd* and *tempch*. Given this relationship, how many equivalence classes should we start out with in the first step of the procedure for partitioning the input domain into equivalence classes?

3.13 (a) Identify weaknesses, as many as you can, of the test T derived in Example 3.10.

(b) Derive a test set that covers each individual equivalence class derived for the four variables in Example 3.8 while ensuring that the semantic relations between different variables are maintained.

(c) Compare the test set derived in (b) with that in Table 3.3 in terms of their respective sizes and error detection effectiveness. If you believe that the error detection effectiveness of the test set you derived is less than that of the test set in Table 3.3, then offer an example of an error in the boiler control software that will likely be not detected by your test set but will likely be detected by the test set in Table 3.3.

3.14 An object named *compute* takes an integer x as input. It is required to send a message to another object O_1 if $x \leq 0$ and a message to object O_2 if $x > 0$. However, due to an error in *compute*, a message is sent to to O_1 when $x < 0$ and to O_2 otherwise. Under what condition(s) will the input $x = 0$ *not* reveal the error in *compute*?

3.15 For each test $t \in T$ in Example 3.12, construct one example of a fault in *textSearch* that is guaranteed to be detected only by t. *Hint:* Avoid trivial examples!

3.16 A method named *cC* takes three inputs: *from*, *to*, and *amount*. Both *from* and *to* are strings and denote the name of a country. Variable *amount* is of type `float`. Method *cC* converts *amount* in the currency of the country specified by *from* and returns the equivalent amount, a quantity of type `float`, in the currency of the country specified by *to*. Here is an example prepared on July 26, 2004:

Inputs: *from* = "USA", *to* = "Japan", *amount* = 100
Returned value: 11,012.0

(a) Derive a set of tests for *cC* using equivalence partitioning and boundary values analysis. (b) Suppose that a GUI encapsulates *cC* and allows the user to select the values of *from* and *to* using a palette of country names. The user types in the amount to be converted in a text box before clicking on the button labeled `Convert`. Will the presence of the GUI change the tests you derived in (a)? If so, how? If not, why?

You may find that the requirement specification given for *cC* is incomplete in several respects. While generating tests, it is recommended that you resolve any ambiguities in the requirements and complete the information not available by using common sense and/or discussing the problem with appropriate members of the design/development team.

3.17 Recall the boundary value analysis in Example 3.11. (a) Construct one example of a boundary error in *fP* that may go undetected unless tests t_2 and t_5 are replaced so that computation at the boundaries of *code* and *qty* is checked in separate tests. (b) Replace t_2 and t_5 by an appropriate set of tests that test *fP* at the boundaries of *code* and *qty* in separate tests.

3.18 Compare test generation using boundary value analysis and equivalence partitioning methods described in this chapter in terms of the number of tests generated, the error detection ability, and the source of tests.

3.19 Consider the following problem for computing the monthly utility bill as formulated by Rob Hierons to illustrate coincidental correctness.

Let W and E denote, respectively, the monthly water and electricity consumption in standard units. Let C_w and C_e denote, respectively, the costs of consuming one standard unit of water and electricity. The monthly charge to the customer is computed as: $(C_w * W + C_e * E)$. However, in situations where a customer uses at least M_w units of water, a 20% discount is applied to the electricity portion of the charge. In this case, the charge to the customer is computed as $(C_w * W + 0.2 * C_e * E)$.

Now suppose that $M_w = 30$ but while coding the computation of the monthly utility charge, the programmer has used by mistake $M_w = 40$. Thus, instead of using the condition $W \geq 30$ the programmer uses the condition $W \geq 40$ to select one of the two formulae above to compute the monthly utility charges. Assume that the function under test (in Java) is as given below.

```
public double incorrectHierons(double w, double e, double cw,
double ce){
  final double mw=40; // Should be 30.
  final double discount=0.2;
  if(w>=mw){
    return(w*cw+discount*ce*e);
  }else{
  return(w*cw+ce*e);
  }
}
```

(a) Use boundary value analysis to determine test inputs to test the (incorrect) function to compute the utility bill. (b) Use partition testing techniques to derive the test cases to test the (incorrect) function to compute the utility bill. (c) Under what conditions will your test fail to discover the error?

3.20 In Example 3.15, suppose that *fP* is to be used in a checkout counter. The cash register at the counter is connected to a central database from where *fP* obtains the various attributes of an item such as name and price. Does this use of *fP* in a networked environment add to the list of environment objects? Can you think of any additional categories and different partitions?

3.21 The coupon field was ignored in the test specification derived in Example 3.17. Given the specifications on page 145, modify the test specifications in Example 3.17 so that correct tests are generated for the processing of the coupon field.

4

Predicate Analysis

CONTENTS

4.1 Introduction
4.2 Domain testing
4.3 Cause-effect graphing
4.4 Tests using predicate syntax
4.5 Tests using basis paths
4.6 Scenarios and tests

The purpose of this chapter is to introduce techniques for the generation of test data from requirements specified using predicates. Such predicates are extracted either from the informally or formally specified requirements or directly from the program under test. There exist a variety of techniques to use one or more predicates as input and generate tests. Some of these techniques can be automated while others may require significant manual effort for large applications. It is not possible to categorize such techniques as either black or white box technique. Depending on how the predicates are extracted, a technique could be termed as a black or white box technique.

4.1 Introduction

The focus of this chapter is the generation of tests from informal as well as rigorously specified requirements. These requirements serve as a source for the identification of the input domain of the application to be developed. A variety of test generation techniques are available to select a subset of the input domain to serve as test set against which the application will be tested.

Figure 3.1 lists some techniques described in this chapter. The figure shows requirements specification in three forms: informal, rigorous, and formal. The input domain is derived from the informal and rigorous specifications. The input domain then serves as a source for test selection. Various techniques, listed in the figure, are used to select a relatively small number of test cases from a usually very large input domain.

4.2 Domain Testing

Path domains and domain errors have been introduced in Chapter 2. Recall that a program contains a domain error if on an input it follows an incorrect path due to incorrect condition or incorrect computation. Domain testing is a technique for generating tests to detect domain errors. The technique is similar to boundary value analysis discussed earlier. However in domain testing, the requirements for test generation are extracted from the code. Hence domain testing could also be considered as a white-box testing technique. While domain testing is also an input domain partitioning technique, it is included in this chapter due to its focus on predicate analysis.

Domain testing aims at selecting tests from the input domain to detect any domain errors. A domain error is said to occur when an incorrect path leads to an incorrect or a correct path leads an incorrect output.

4.2.1 Domain errors

As mentioned earlier, a path condition consists of a set of predicates that occur along the path in a conditional or a loop statement. A path condition gives rise to a path domain that has a boundary. The boundary consists of one or more border segments, also known as *borders*. There is one border corresponding to each predicate in the path condition. An error in any of the predicates in a path condition leads to a change in the boundary due to *border shift*. It is such border shifts, probably unintended, the determination of which is the target of domain testing.

Every path in a program has a condition associated with it. This condition is known as a path condition and must hold for the path to be traversed.

Considering only linear predicates without any Boolean conditions such as AND and OR, the boundary and its borders can be shown as in Figure 4.1. The figure shows three types of domain errors corresponding to the correct path domain in Figure 4.1(a). A border shift occurs as shown in

Figure 4.1 Types of domain errors. (a) Correct boundary corresponding to the path condition $y < x + 1 \wedge y > 0 \wedge x < 0$. (b) and (c) Boundary shifted due to error in $y < x + 1$. (d) Extra border due to the inclusion of $y < 0.5$ in the path condition. (e) Missing border due to missing $y < x + 1$. The dots indicate points inside the boundary.

Figure 4.1(b) and (c). The border shift in Figure 4.1(b) is parallel whereas in Figure 4.1(c) it is tilted. The method described here does not depend on the kind of border shift. Figure 4.1(d) shows an extra border corresponding to the condition $y < 0.5$. Figure 4.1(e) shows a missing border due to a missing condition $y < x + 1$.

4.2.2 Border shifts

A border shift occurs due to an incorrect condition. The shift refers to a change in the boundary of path domain due to an error in the coding of the path condition.

As mentioned earlier, and illustrated in Figure 4.1, a domain error occurs due to a possibly unintended shift in the border. Thus, the idea underlying domain testing is to sample points inside and outside the displaced area in the hope that the error would be detected. This idea is independent of whether or not a border is linear or non-linear.

As we now know, in the case of linear constraints in two dimensions, a path domain is specified by a set of one or more straight line borders. A border is a line corresponding to a simple predicate. A border that corresponds to a predicate using any of the operators $\leq, \geq$, and $=$, is considered *closed*. Thus, points that lie on a closed border are part of the path domain. A border that corresponds to a predicate using any of the operators $<, >$ or $\neq$ is considered *open*. Points that lie on the open border are not part of the path domain.

Note that a border is always defined using equality. However, whether or not points defined by a predicate lie in or out of the path domain is determined by the actual relational operator in the predicate. For example, the predicate $y - x < 0$ defines an open border. Hence all points that satisfy the relation $y - x = 0$ are not included in the path domain while those that satisfy the inequality are. However, for a predicate $y - x \leq 0$, all points that satisfy the equality are included in the path domain.

4.2.3 ON-OFF points

Testing for boundary shifts is done with the help of ON and OFF points. An ON point must satisfy the path condition associated with the border. It need not be exactly on the given border but should be near. An OFF point must be as close as possible to the ON point for that border. It lies outside the border and hence does not satisfy the condition associated with the border.

An ON point occurs on or near a boundary. An OFF point is close to an OFF point but lies outside the border and does not satisfy the path condition.

While testing for a boundary shift error, only one ON and one OFF point is needed for inequality constraints. For an equality (=) or non-equality (≠) constraint one ON point and two OFF points are selected; the OFF points lie on the opposite sides of the border. The following examples illustrate the selection of ON and OFF points and the detection of domain errors.

The number of ON and OFF points needed to test a path condition depends on whether an equality or an inequality constraint occurs in the condition.

Example 4.1 Consider the following contrived function f that we assume to be incorrect. The two domain errors in f are indicated alongside the erroneous statements. Depending on the inputs f displays a string consisting of a sequence of integers from 1 through 4. The displayed sequence depends on the values of its input parameters x and y.

```
 1 String f(int x, int y){
 2   String s=" ";
 3   if(y<x+1) // Error 1: this should be y<x+2
 4     s=s+"1";
 5   if(y>0)
 6     s=s+"2";
 7   if(x<=0) // Error 2:  this should be x<y
 8     s=s+"3";
 9   s=s+"4";
10   return s;
11 }
```

For the sake of simplicity let us consider the following path p through function f: 1, 2, 3, 4, 5, 6, 7 , 9, 10. The path condition

c is $y < x + 1 \wedge y > 0 \wedge x \leq 0$. Given that f has two errors, the correct path condition c' is $y < x + 2 \wedge y > 0 \wedge x < y$. Figure 4.2(a) shows the path domains corresponding to c and c'. The shaded area in the figure denotes the incorrect path domain while the dashed area the correct one. Generation of tests for the path condition proceeds as follows.

ON and OFF points are generated for each simple predicate in a path condition. It is best to avoid the reuse of points.

1. Generate ON and OFF points for each predicate in the path condition. It is best to avoid the reuse of points to increase chances of finding errors.
2. For each point determine the expected value of the program under test. Each point, together with the expected output, constitutes a test case.
3. Run the program under test on each test case and determine if the actual output is the same as the expected output.

Figure 4.2(b) shows a scaled version of the incorrect path domain and the corresponding ON and OFF points. Given that x and y are integers, the shaded area is empty in the sense that there are no points inside this area.

The following table lists the outcome of generating the ON and OFF points and executing f against these tests. In the table f denotes the function under test and f' the correct function. Obviously, f' is not known during testing. However, we assume it is possible to determine the expected output $f'(x, y)$, from the requirements.

Figure 4.2 (a) Path domains corresponding to the incorrect and correct path conditions in Example 4.1. (b) Incorrect path domain, scaled up, and the ON-OFF points.

Condition	Point	Test			c	$f(x, y)$	$f'(x, y)$	Error
		#	x	y				
$y < x + 1$	ON	1	1	1	true	"124"	"124"	Not detected
	OFF	2	0	1	false	"234"	"1234"	Detected
$y > 0$	ON	3	1*	1	true	"124"	"124"	No error
	OFF	4	1*	−1	false	"14"	"14"	No error
$x \leq 0$	ON	5	0	2*	true	"234"	"234"	Not detected
	OFF	6	1	2*	false	"24"	"1234"	Detected

*Any value will satisfy the requirement imposed by the point type.

The rightmost three columns in the above table show the outcome of executing f against the four tests and the expected outcome. Note that in this case both errors are detected by the OFF points.

The above example is simple and used for illustrating how tests are determined using the ON and OFF points. Notice that all conditions along a path are independent in the sense that the outcome of a condition does not depend on that of any other condition. In practice, this may not be true. Exercise 4.1 illustrates dependent predicates in a path condition.

A path condition might contain simple conditions that depend on each other. In such cases it is best to simplify the path condition prior to the generation f ON and OFF points so that the revised path condition consists of independent simple conditions. Of course the original and the new conditions must be semantically equivalent.

Example 4.2 Suppose that the predicate $x == y$ defines a border in a path domain for integer inputs x and y. This is an equality border and we need one ON and two OFF points. Following are the test cases for testing any border shift corresponding to this border.

ON: $(x = 1, y = 1)$
OFF1: $(x = 1, y = 2)$
OFF2: $(x = 2, y = 1)$

Note that the two OFF points are on the opposite sides of the border and close to the ON point. Now suppose that the correct border is defined by the condition $x == y + 1$. In this case, the test cases corresponding to the ON and OFF2 points lead to different outcomes for the two conditions. For any other equality, the ON point leads to different outcomes for the two conditions.

4.2.4 Undetected errors

Domain testing as described above is not guaranteed to detect errors due to shifted boundaries. An error might go undetected when the correct

boundary lies between the incorrect boundary and the OFF point. The next example illustrates such a situation.

An error might go undetected when the correct boundary lies between the two incorrect boundaries.

Example 4.3 Suppose that the incorrect path condition in the following function is $x < y$. The correct condition is assumed to be $x < y + 1$.

```
1
2 int f(double x, double y){
3   if(x<y) // Correct condition is x<y+1
4     return 1;
5   else
6     return 2;
7 }
```

One possible set of ON and OFF points derived from the incorrect condition are: ON: $(x = 1, y = 1.001)$ and OFF: $(x = 1.5, y = 0.4)$. Note that, as required, the ON point satisfies the border condition and is close to the border. Again, as required, the OFF point does not satisfy the border condition. However, the OFF point is not close to the ON point.

For both test cases, the correct and the incorrect path conditions evaluate to the same truth values and hence the error goes undetected. As shown in Figure 4.3, the correct border corresponding to $x < y + 1$ passes between the incorrect border and the OFF point.

Now suppose we select the OFF point very near the incorrect border, as it should have been. Thus, for example, OFF: $(x = 1,$

Figure 4.3 Path domains corresponding $x < y$ (incorrect) and $x < y + 1$ (correct). ON and OFF points shown do not detect the error. Note that the correct border lies between the incorrect border and the OFF point. OFF′ point does detect the error as it lies between the incorrect and the correct borders (see Example 4.3.)

y = 0.999). In this case, the OFF point lies between the correct and the incorrect borders. The error is now detected because the incorrect condition evaluates to false but the correct condition evaluates to true.

4.2.5 Coincidental correctness

Coincidental correctness may also cause an error to go undetected. This happens when the expected and the actual outcomes of executing a program match despite a program error.

Coincidental correctness occurs when the expected and the actual output of a program match despite the execution of an incorrect path.

Example 4.4 Consider the following function.

```
1 int f(int z){
2   if(z>15) // Error: The condition should be z>=15.
3     return z-10;
4   else
5     return z%10;
6   }
```

Now suppose that the function above is invoked with the value of z as 15. In this case, the function would return 5 regardless of whether the path condition is as given or the correct one. Thus, the error is not revealed. Notice that the error is revealed by several other values of z, for example, for $z = 5$. Also see Exercise 4.19 for another example.

4.2.6 Paths to be tested

The generation of tests using domain testing starts with the selection of a path. The path selected then determines the predicates that lie on it and hence the path domain. A tester may select a path based on a variety of criteria. For example, there might be a path that is considered critical to the application under test. This could become a selected path.

Most programs have an exorbitantly large number of paths. A tester needs to decide one or more suitable criteria to select which of the many paths are to be tested.

In general, one needs to pick a suitable criterion to decide what paths to select. Several useful criteria are introduced in Chapter 7. As an example, one could use the boundary-interior coverage criterion. The selected paths must start from the beginning of the program and ensure that each loop is executed zero times along one path and once along the other. The execution of the body zero times corresponds to the boundary condition. The execution of the body once corresponds to the interior condition. In some cases only one selected path might satisfy the condition though it is likely that at least two will be needed.

4.3 Cause-Effect Graphing

We described two techniques for test selection based on equivalence partitioning and boundary value analysis. One is based on unidimensional partitioning and the other on multidimensional partitioning of the input domain. While equivalence classes created using multidimensional partitioning allow the selection of a large number of input combinations, the number of such combinations could be astronomical. Further, many of these combinations, as in Example 4.8, are infeasible and make the test selection process tedious.

Cause-effect graphing is a visual technique for modeling the relationships between inputs and output of a program. The inputs (cases) could be specified as conditions and the output (effects) in terms of expected values or behavior.

Cause-effect graphing, also known as *dependency modeling*, focuses on modeling dependency relationships amongst program input conditions, known as *causes*, and output conditions, known as *effects*. The relationship is expressed visually in terms of a cause-effect graph. The graph is a visual representation of a logical relationship amongst inputs and outputs that can be expressed as a Boolean expression. The graph allows selection of various combinations of input values as tests. The combinatorial explosion in the number of tests is avoided by using certain heuristics during test generation.

A cause is any condition in the requirements that may effect the program output. An effect is the response of the program to some combination of input conditions. It may be, for example, an error message displayed on the screen, a new window displayed, or a database updated. An effect need not be an "output" visible to the user of the program. Instead, it could also be an internal *test point* in the program that can be probed during testing to check if some intermediate result is as expected. For example, the intermediate test point could be at the entrance into a method to indicate that indeed the method has been invoked.

A requirement such as "Dispense food only when the DF switch is ON" contains a cause "DF switch is ON" and an effect "Dispense food." This requirement implies a relationship between the "DF switch is ON" and the effect "Dispense food." Of course, other requirements might require additional causes for the occurrence of the "Dispense food" effect. The following generic procedure is used for the generation of tests using cause-effect graphing.

1. Identify causes and effects by reading the requirements. Each cause and effect is assigned a unique identifier. Note that an effect can also be a cause for some other effect.
2. Express the relationship between causes and effects using a cause-effect graph.

3. Transform the cause-effect graph into a limited entry decision table, hereafter referred to simply as decision table.
4. Generate tests from the decision table.

The basic notation used in cause-effect graphing and a few illustrative examples follow.

4.3.1 Notation used in cause-effect graphing

The basic elements of a cause-effect graph are shown in Figure 4.4. A typical cause-effect graph is formed by combining the basic elements so as to capture the relations between causes and effects derived from the requirements. The semantics of the four basic elements shown in Figure 4.4 is expressed below in terms of the if-then construct; C, C_1, C_2, and C_3 denote causes and Ef denotes an effect.

C implies Ef :	if (C) then Ef;
not C implies Ef :	if ($\neg C$) then Ef;
Ef when C_1 and C_2 and C_3 :	if (C_1 && C_2 && C_3) then Ef;
Ef when C_1 or C_2 :	if ($C_1 \parallel C_2$) then Ef;

There often arise constraints amongst causes. For example, consider an inventory control system that tracks the inventory of various items that are stocked. For each item, an inventory attribute is set to "Normal," "Low," and "Empty." The inventory control software takes actions as the value of this attribute changes. When identifying causes, each of the three inventory levels will lead to a different cause listed below.

Constraints among conditions (effects) can be specified in a cause-effect graph.

Figure 4.4 Basic elements of a cause-effect graph: *implication*, *not* ($\sim$), *and* ($\wedge$), *or* ($\vee$). C, C_1, C_2, and C_3 denote causes and Ef denotes effect. An arc is used, for example in the *and* relationship, to group three or more causes.

Constraints among causes are E (Exclusive), I (inclusive), R (required), and O (one and only one). These constraints, except R, are n-ary meaning that they hold among two or more causes. The R constraint is considered binary.

Figure 4.5 Constraints amongst causes (*E*, *I*, *O*, and *R*).

C_1 : "Inventory is normal"
C_2 : "Inventory is low"
C_3 : "Inventory is zero"

However, at any instant, exactly one of C_1, C_2, C_3 can be true. This relationship amongst the three causes is expressed in a cause-effect graph using the "Exclusive (E)" constraint shown in Figure 4.5. In addition, shown in this figure are the "Inclusive (I)," "Requires (R)," and "One and only one (O)," constraints. The I constraint between two causes C_1 and C_2 implies that at least one of the two must be present. The R constraint between C_1 and C_2 implies that C_1 requires C_2. The O constraint models the condition that one, and only one, of C_1 and C_2 must hold.

The table below lists all possible values of causes constrained by E, I, R, and O. A 0 or a 1 under a cause implies that the corresponding condition is, respectively, false and true. The arity of all constraints, except R, is greater than 1, i.e., all except the R constraint can be applied to two or more causes; the R constraint is applied to two causes.

Constraint	**Arity**	**Possible values**		
		C_1	C_2	C_3
$E(C_1, C_2, C_3)$	$n \geq 2$	0	0	0
		1	0	0
		0	1	0
		0	0	1
$I(C_1, C_2)$	$n \geq 2$	1	0	–
		0	1	–
		1	1	–

Constraint	Arity	Possible values		
		C_1	C_2	C_3
$R(C_1, C_2)$	$n = 2$	1	1	–
		0	0	–
		0	1	–
$O(C_1, C_2, C_3)$	$n \geq 2$	1	0	0
		0	1	0
		0	0	1

Figure 4.6 The masking constraint amongst effects.

In addition to the constraints on causes, there could also be constraints on effects. The cause-effect graphing technique offers one constraint, known as "Masking (M)" on effects. Figure 4.6 shows the graphical notation for the masking constraint. Consider the following two effects in the inventory example mentioned earlier.

Constraints could also be specified among the effects. One such constraint is M (masking) that indicates that a given effect masks another.

Ef_1 : Generate "Shipping invoice."
Ef_2 : Generate an "Order not shipped" regret letter.

Effect Ef_1 occurs when an order can be met from the inventory. Effect Ef_2 occurs when the order placed cannot be met from the inventory or when the ordered item has been discontinued after the order was placed. However, Ef_2 is masked by Ef_1 for the same order, i.e. both effects cannot occur for the same order.

A condition that is false (true) is said to be in the "0 state" (1 state). Similarly, an effect can be "present" (1 state) or "absent" (0 state).

4.3.2 Creating cause-effect graphs

The process of creating a cause-effect graph consists of two major steps. First, the causes and effects are identified by a careful examination of the requirements. This process also exposes the relationships amongst various causes and effects as well as constraints amongst the causes and effects.

Each cause and effect is assigned a unique identifier for ease of reference in the cause-effect graph.

A cause-effect graph is derived from causes and effects derived from the requirements.

In the second step, the cause-effect graph is constructed to express the relationships extracted from the requirements. When the number of causes and effects is large, say over 100 causes and 45 effects, it is appropriate to use an incremental approach. An illustrative example follows.

Example 4.5 Let us consider the task of test generation for a GUI based computer purchase system. A web-based company is selling computers (CPU), printers (PR), monitors (M), and additional memory (RAM). An order configuration consists of one to four items as shown in Figure 4.7. The GUI consists of four windows for displaying selections from CPU, Printer, Monitor, and RAM and one window where any free giveaway items are displayed.

For each order, the buyer may select from three CPU models, two printer models, and three monitors. There are separate windows one each for CPU, printer, and monitor that show the possible selections. For simplicity we assume that RAM is available only as an upgrade and that only one unit of each item can be purchased in one order.

Monitors M 20 and M 23 can be purchased with any CPU or as a stand-alone item. M 30 can only be purchased with CPU 3. PR 1 is available free with the purchase of CPU 2 or CPU 3. Monitors and printers, except for M 30, can also be purchased separately without purchasing any CPU. Purchase of CPU 1 gets RAM 256 upgrade, purchase of CPU 2 or CPU 3 gets a RAM 512 upgrade. The RAM 1G upgrade and a free PR 2 is available when CPU 3 is purchased with monitor M 30.

When a buyer selects a CPU, the contents of the printer and monitor windows are updated. Similarly, if a printer or a monitor is selected, contents of various windows are updated. Any free printer and RAM available with the CPU selection is displayed in a different window marked "Free." The total price, including taxes,

$$\begin{Bmatrix} \text{CPU 1} \\ \text{CPU 2} \\ \text{CPU 3} \end{Bmatrix} \quad \begin{Bmatrix} \text{PR 1} \\ \text{PR 2} \end{Bmatrix} \quad \begin{Bmatrix} \text{M 20} \\ \text{M 23} \\ \text{M 30} \end{Bmatrix} \quad \begin{Bmatrix} \text{RAM 256} \\ \text{RAM 512} \\ \text{RAM 1G} \end{Bmatrix}$$

Figure 4.7 Possible configurations of a computer system sold by a web-based company. CPU: CPU configuration, PR: Printer, M: Monitor. RAM: memory upgrade.

for the items purchased is calculated and displayed in the "Price" window. Selection of a monitor could also change the items displayed in the "Free" window. Sample configurations and contents of the "Free" window are given below.

Items purchased	"Free" window	Price
CPU 1	RAM 256	$499
CPU 1. PR 1	RAM 256	$628
CPU 2. PR 2. M23	PR 1, RAM 512	$2257
CPU 3. M30	PR 2, RAM 1G	$3548

The first step in cause-effect graphing is to read the requirements carefully and make a list of causes and effects. For this example, we will consider only a subset of the effects and leave the remaining as an exercise. This strategy also illustrates how one could generate tests using an incremental strategy.

A cause in a cause-effect graph could be a condition such as "Purchase an item" or "Deposit money". An effect could be, for example, "Print invoice" or "Update account balance." An effect could become a cause for another effect.

A careful reading of the requirements is used to extract the following causes. We have assigned a unique identifier, C_1 through C_8 to each cause. Each cause listed below is a condition that can be true or false. For example, C_8 is true if monitor M 30 is purchased.

C_1 : Purchase CPU 1.
C_2 : Purchase CPU 2.
C_3 : Purchase CPU 3.
C_4 : Purchase PR 1.
C_5 : Purchase PR 2.
C_6 : Purchase M 20.
C_7 : Purchase M 23.
C_8 : Purchase M 30.

Note that while it is possible to order any of the items listed above, the GUI will update the selection available depending on which CPU, or any other item, is selected. For example, if CPU 3 is selected for purchase then monitors M 20 and M 23 will not be available in the monitor selection window. Similarly, if monitor M 30 is selected for purchase, then CPU 1 and CPU 2 will not be available in the CPU window.

Next, we identify the effects. In this example, the application software calculates and displays the list of items available free with the purchase and the total price. Hence the effect is in terms of the

contents of the "Free" and "Price" windows. There are several other effects related to the GUI update actions. These effects are left for the exercise (see Exercise 4.4).

Calculation of the total purchase price depends on the items purchased and the unit price of each item. The unit price is obtained by the application from a price database. The price calculation and display is a cause that creates the effect of displaying the total price. For simplicity, we ignore the price related cause and effect. The set of effects in terms of the contents of the "Free" display window are listed below.

Ef_1 : RAM 256.
Ef_2 : RAM 512 and PR 1.
Ef_3 : RAM 1G and PR 2.
Ef_4 : No giveaway with this item.

Once the conditions (causes) and the expected outcome (effects) have been identified, the relationship amongst them can be expressed using a cause-effect graph.

Now that the causes, effects, and their relationships have been identified, we can begin to construct the cause-effect graph. Figure 4.8 shows the complete graph that expresses the relationships between C_1 through C_8 and effects Ef_1 through Ef_4.

From the cause-effect graph in Figure 4.8 we notice that C_1, C_2, and C_3 are constrained using the E (exclusive) relationship. This expresses the requirement that only one CPU can be purchased in one order. Similarly, C_3 and C_8 are related via the R (requires) constraint to express the requirement that monitor M 30 can only be purchased with CPU 3. Relationships amongst causes and effects are expressed using the basic elements shown earlier in Figure 4.4.

Notice the use of an intermediate node labeled 1 in Figure 4.8. Though not necessary in this example, such intermediate nodes are often useful when an effect depends on conditions combined using more than one operator, for example, $(C_1 \wedge C_2) \vee C_3$. Note also that purchase of printers and monitors without any CPU leads to no free item (Ef_4).

The relationships between effects and causes shown in Figure 4.8 can be expressed in terms of Boolean expressions as follows:

$$Ef_1 = C_1$$
$$Ef_2 = C_2 \vee C_3$$
$$Ef_3 = C_3 \wedge C_8$$
$$Ef_4 = C_4 \wedge C_5 \wedge C_6 \wedge C_7$$

Figure 4.8 Cause-effect graph for the web-based computer sale application. C_1, C_2, and C_3 denote the purchase of, respectively, CPU 1, CPU 2, and CPU 3. C_4 and C_5 denote the purchase of printers PR 1 and PR 2, respectively. C_6, C_7, and C_8 denote the purchase of monitors M 20, M 23, and M 30, respectively.

4.3.3 Decision table from cause-effect graph

We will now see how to construct a decision table from a cause-effect graph. Each column of the decision table represents a combination of input values, and hence a test. There is one row for each condition and effect. Thus the decision table can be viewed as an $N \times M$ matrix with N being the sum of the number of conditions and effects and M the number of tests.

A decision table is a tabular representation of a cause-effect graph. It makes it easy to generate tests.

Each entry in the decision table is a 0 or a 1 depending on whether or not the corresponding condition is false or true, respectively. For a row corresponding to an effect, an entry is 0 or 1 if the effect is not present or present, respectively. Following is a procedure to generate a decision table from a cause-effect graph.

`Procedure for generating a decision table from a cause-effect graph.`

Input: A cause-effect graph containing causes $C_1, C_2, \ldots, C_p$ and effects $Ef_1, Ef_2, \ldots, Ef_q$.

Output: A decision table DT containing $N = p + q$ rows and M columns, where M depends on the relationship between the causes and effects as captured in the cause-effect graph.

Procedure: `CEGDT`

/* i is the index of the next effect to be considered.

next_dt_col is the next empty column in the decision table.

V_k: a vector of size $p + q$ containing 1's and 0's. $V_j, 1 \leq j \leq p$, indicates the state of condition C_j and $V_l, p < l \leq p + q$, indicates the presence or absence of effect Ef_{l-p}.

*/

Step 1 *Initialize DT to an empty decision able.*

$next_dt_col = 1$.

Step 2 Execute the following steps for $i = 1$ to q.

2.1 *Select the next effect to be processed.*

Let $e = Ef_i$.

Determination of a combination of conditions that lead to the presence of an effect is done by tracing backwards in the cause-effect starting at an effect and stopping when the truth values of all relevant conditions have been determined.

2.2 *Find combinations of conditions that cause e to be present.*

Assume that e is present. Starting at e, trace the cause-effect graph backwards and determine the combinations of conditions $C_1, C_2, \ldots, C_p$ that lead to e being present. Avoid combinatorial explosion by using the heuristics given in the text following this procedure. Make sure that the combinations satisfy any constraints amongst the causes.

Let $V_1, V_2, \ldots, V_{m_i}$ be the combinations of causes that lead to e being present. There must be at least one combination that makes e to be present, i.e. in 1 state, and hence $m_i \geq 1$. Set $V_k(l), p < l \leq p + q$ to 0 or 1 depending on whether effect Ef_{l-p} is present or not for the combination of all conditions in V_k.

2.3 *Update the decision table.*

Add $V_1, V_2, \ldots, V_{m_i}$ to the decision table as successive columns starting at *next_dt_col*.

2.4 *Update the next available column in the decision table.*

$next_dt_col = next_dt_col + m_i$. At the end of this procedure, $next_dt_col - 1$ is the number of tests generated.

`End of Procedure CEGDT`

Procedure CEGDT can be automated for the generation of tests. However, as indicated in Step 2.2, one needs to use some heuristics in

order to avoid combinatorial explosion in the number of tests generated. Before we introduce the heuristics, we illustrate through a simple example the application of procedure CEGDT *without* applying the heuristics in Step 2.

Example 4.6 Consider the cause-effect graph in Figure 4.9. It shows four causes labeled C_1, C_2, C_3, and C_4 and two effects labeled Ef_1 and Ef_2. There are three intermediate nodes labeled 1, 2, and 3. Let us now follow procedure CEGDT step-by-step to generate a decision table.

In Step 1 we set $next_dt_col = 1$ to initialize the decision table to empty. Next, $i = 1$ and, in accordance with Step 2.1, $e = \mathrm{Ef}_1$. Continuing further and in accordance with Step 2.2, trace backwards from e to determine combinations that will cause e to be present. e must be present when node 2 is in 1-state. Moving backwards from node 2 in the cause-effect graph, note that any of the following three combinations of states of nodes 1 and C_3 will lead to e being present: (0, 1), (1, 1), and (0, 0).

Node 1 is also an internal node and hence move further back to obtain the values of C_1 and C_2 that effect node 1. Combination of C_1 and C_2 that brings node 1 to the 1-state is (1, 1) and combinations that bring it to 0-state are (1, 0), (0, 1), and (0, 0). Combining this information with that derived earlier for nodes 1 and C_3, we obtain the following seven combinations of C_1, C_2, and C_3 that cause e to be present.

1	0	1
0	1	1
0	0	1

Figure 4.9 A cause-effect graph to illustrate procedure CEGDT.

1 1 1
1 0 0
0 1 0
0 0 0

Next, from Figure 4.9, note that C_3 requires C_1 which implies that C_1 must be in 1-state for C_3 to be in 1-state. This constraint makes infeasible the second and third combinations above. In the end, we obtain the following five combinations of the four causes that lead to e being present.

1 0 1
1 1 1
1 0 0
0 1 0
0 0 0

Setting C_4 to 0 and appending the values of Ef_1 and Ef_2, we obtain the following five vectors. Note that $m_1 = 5$ in Step 2. This completes the application of Step 2.2, without the application of any heuristics, in the CEGDT procedure.

V_1 1 0 1 0 1 0
V_2 1 1 1 0 1 0
V_3 1 0 0 0 1 0
V_4 0 1 0 0 1 0
V_5 0 0 0 0 1 0

The five vectors are transposed and added to the decision table starting at column *next_dt_col* which is 1. The decision table at the end of Step 2.3 follows.

	1	2	3	4	5
C_1	1	1	1	0	0
C_2	0	1	0	1	0
C_3	1	1	0	0	0
C_4	0	0	0	0	0
Ef_1	1	1	1	1	1
Ef_2	0	0	0	0	0

We update *next_dt_col* to 6, increment i to 2 and get back to Step 2.1. We now have $e = \mathrm{Ef}_2$. Tracing backwards, we find that for e to be present, node 3 must be in the 1-state. This is possible with only one combination of node 2 and C_4, which is (1, 1).

Earlier we derived the combinations of C_1, C_2, and C_3 that lead node 2 into the 1-state. Combining these with the value of C_4 we arrive at the following combination of causes that lead to the presence of Ef_2.

1 0 1 1
1 1 1 1
1 0 0 1
0 1 0 1
0 0 0 1

From Figure 4.9 we note that C_2 and C_4 cannot be present simultaneously. Hence, we discard the second and the fourth combinations from the list above and obtain the following three feasible combinations.

1 0 1 1
1 0 0 1
0 0 0 1

Appending the corresponding values of Ef_1 and Ef_2 to each of the above combinations, we obtain the following three vectors.

V_1 1 0 1 1 1 1
V_2 1 0 0 1 1 1
V_3 0 0 0 1 1 1

Transposing the vectors listed above and appending them as three columns to the existing decision table, we obtain the following.

	1	2	3	4	5	6	7	8
C_1	1	1	1	0	0	1	1	0
C_2	0	1	0	1	0	0	0	0
C_3	1	1	0	0	0	1	0	0
C_4	0	0	0	0	0	1	1	1
Ef_1	1	1	1	1	1	1	1	1
Ef_2	0	0	0	0	0	1	1	1

Next we update *nxt_dt_col* to 9. Of course, doing so is useless as the loop set up in Step 2 is now terminated. The decision table listed above is the output obtained by applying procedure CEGDT to the cause-effect graph in Figure 4.9.

4.3.4 Heuristics to avoid combinatorial explosion

A brute force method to determine conditions that will cause an effect to be present will likely generate a large number of tests. A few simple heuristics can be used to avoid a combinatorial explosion.

While the application of heuristics will likely reduce the number of tests, it might also reduce the error detection effectiveness of the generated test suite.

While tracing back through a cause-effect graph we generate combinations of causes that set an intermediate node, or an effect, to a 0 or a 1 state. Doing so in a brute force manner could lead to a large number of combinations. In the worst case, if n causes are related to an effect e, then the maximum number of combinations that bring e to a 1-state is 2^n.

As tests are derived from the combinations of causes, large values of n could lead to an exorbitantly large number of tests. We avoid such a combinatorial explosion by using simple heuristics related to the "AND" ($\wedge$) and "OR" ($\vee$) nodes.

Certainly, the heuristics described below are based on the assumption that certain types of errors are less likely to occur than others. Thus, while applying the heuristics to generate test inputs will likely lead to a significant reduction in the number of tests generated, it may also discard tests that would have revealed a program error. Hence, one must apply the heuristics with care and only when the number of tests generated without their application is too large to be useful in practice.

A set of four heuristics labeled H_1 through H_4, is given in Table 4.1. The leftmost column shows the node type in the cause-effect graph, the center column is the desired state of the dependent node, and the rightmost column is the heuristic for generating combinations of inputs to the nodes that effect the dependent node e.

For simplicity we have shown only two nodes n_1 and n_2 and the corresponding effect e; in general there could be one or more nodes related to e. Also, each of n_1 and n_2 might represent a cause or may be an internal node with inputs, shown as dashed lines, from causes or other internal nodes. The next example illustrates the application of the four heuristics shown in Table 4.1.

Example 4.7 Consider the cause-effect graph in Figure 4.10. We have at least two choices while tracing backwards to derive the necessary combinations: derive all combinations first and then apply the heuristics and derive combinations while applying the heuristics. Let us opt for the first alternative as it is a bit simple to use in this example.

Suppose that we require node e to be 1. Tracing backwards, this requirement implies that node a must be a 0. If we trace backwards further, without applying any heuristic, we obtain the following seven combinations of causes that bring e to 1 state. The last column lists the inputs to node a. The combinations that correspond

Table 4.1 Heuristics used during the generation of input combinations from a cause-effect graph.

Node type	Desired state of e	Input combinations
	0	H_1: Enumerate all combinations of inputs to n_1 and n_2 such that $n_1 = n_2 = 0$.
	1	H_2: Enumerate all combinations of inputs to n_1 and n_2 other than those for which $n_1 = n_2 = 0$.
	0	H_3: Enumerate all combinations of inputs to n_1 and n_2 such that each of the possible combinations of n_1 and n_2 appears exactly once and $n_1 = n_2 = 1$ does not appear. Note that for two nodes, as in the figure on the left, there are three such combinations: (0, 0), (0, 1), and (1, 0). In general, for k nodes combined using the logical and operator to form e, there are $2^k - 1$ such combinations.
	1	H_4: Enumerate all combinations of inputs to n_1 and n_2 such that $n_1 = n_2 = 1$.

Figure 4.10 Cause-effect graph for Example 4.7.

One may apply the heuristics while generating the conditions for an effect to be present or not present. Alternately one may first generate the conditions using a brute force method and then apply the heuristics to remove the undesirable conditions as shown here.

to the inputs to node a listed in the rightmost column are separated by horizontal lines.

Let us now generate tests using the heuristics applicable to this example. First we note that node a matches the OR-node shown in the top half of Table 4.1. As we want the state of node a to be 0, heuristic H_1 applies in this situation. H_1 asks us to enumerate all combinations of inputs to node a such that C_1 and node b are 0. (0, 0) is the only such combination and is listed in the last column of the following table.

	C_1	C_2	C_3	Inputs to node a
1	0	0	0	$C_1 = 0, b = 0$
2	0	0	1	
3	0	1	0	
4	0	1	1	$C_1 = 0, b = 1$
5	1	0	0	$C_1 = 1, b = 0$
6	1	0	1	
7	1	1	0	

Let us begin with (0, 0). No heuristic applies to C_1 as it has no preceding nodes. Node b is an AND-node as shown in the bottom half of Table 4.1. We want node b to be 0 and therefore H_3 applies. In accordance with H_3 we generate three combinations of inputs to node b: (0, 0), (0, 1), and (1, 0). Notice that combination (1, 1) is forbidden. Joining these combinations of C_2 and C_3 with $C_1 = 0$, we obtain the first three combinations listed in the preceding table.

Though not required here, suppose that we were to consider the combination $C_1 = 0, b = 1$. Heuristic H_4 applies in this situation. As both C_2 and C_3 are causes with no preceding nodes, the only combination we obtain now is (1, 1). Combining this with $C_1 = 0$ we obtain sequence 4 listed in the preceding table.

We have completed the derivation of combinations using the heuristics listed in Table 4.1. Note that the combinations listed above for $C_1 = 1, b = 0$ are not required. Thus, we have obtained only three combinations instead of the seven enumerated earlier. The reduced set of combinations is listed below.

	C_1	C_2	C_3	Inputs to node a
1	0	0	0	$C_1 = 0, b = 0$
2	0	0	1	
3	0	1	0	

Application of a heuristic may or may not save on tests.

Let us now examine the rationale underlying the various heuristics for reducing the number of combinations. Heuristic H_1 does not save us on any combinations. The only way an OR-node can cause its effect e to be 0 is for all its inputs to be 0. H_1 suggests that we enumerate all such combinations. Heuristic H_2 suggests that we use all combinations that cause e to be 1 except those that cause $n_1 = n_2 = 0$. To understand the rationale underlying

H_2 consider a program required to generate an error message when condition c_1 or c_2 is true. A correct implementation of this requirement is given below.

```
if (c1 ∨ c2) print("Error");
```

Now consider the following erroneous implementation of the same requirement.

```
if (c1 ∨ ¬c2) print("Error");
```

A test that sets both c_1 and c_2 true will not be able to detect an error in the implementation above if short circuit evaluation is used for Boolean expressions. However, a test that sets $c_1 = 0$ and $c_2 = 1$ will be able to detect this error. Hence H_2 saves us from generating all input combinations that generate the pair (1, 1) entering an effect in an OR-node (see Exercise 4.6).

Heuristics H_3 saves us from repeating the combinations of n_1 and n_2. Once again this could save us a lot of tests. The assumption here is that any error in the implementation of e will be detected by tests that cover different combinations of n_1 and n_2. Thus, there is no need to have two or more tests that contain the same combination of n_1 and n_2.

Lastly, H_4 for the AND-node is analogous to H_1 for the OR-node. The only way an AND-node can cause its effect e to be 1 is for all its inputs to be 1. H_4 suggests that we enumerate all such combinations. We stress, once again, that while the heuristics discussed above will likely reduce the set of tests generated using cause-effect graphing, they might also lead to useful tests being discarded. Of course, in general and prior to the start of testing, it is almost impossible to know which of the test cases discarded will be useless and which ones useful.

4.3.5 Test generation from a decision table

Test generation from a decision table is relatively straightforward. Each column in the decision table generates at least one test input. Note that each combination might be able to generate more than one test when a condition in the cause-effect graph can be satisfied in more than one way. For example, consider the following cause:

Tests are generated from the decision table obtain from a cause-effect graph. Each combination of conditions may generate more than one test case.

C: $x < 99$.

The condition above can be satisfied by many values such as $x = 1$, and $x = 49$. Also, C can be made false by many values of x such as $x = 100$ and

$x = 999$. Thus, one might have a choice of values of input variables while generating tests using columns from a decision table.

While one could always select values arbitrarily as long as they satisfy the requirement in the decision table, it is recommended that the choice be made so that tests generated are different from those that may have already been generated using some other technique such as, for example, boundary value analysis. Exercise 4.8 asks you to develop tests for the GUI-based computer purchase system in Example 4.5.

4.4 Tests Using Predicate Syntax

In this section, we introduce techniques for generating tests that are aimed at detecting faults in the coding of conditions. The conditions from which tests are generated might arise from requirements or might be embedded in the program to be tested.

A condition is represented formally as a predicate. For example, consider the requirement "if the printer is ON and has paper then send the document for printing." This statement consists of a condition part and an action part. The following predicate, denoted as p_r, represents the condition part of the statement.

Cause-effect graphing is a visual method for capturing the requirements and then generating tests. Another way to generate tests from predicates, which are conditions in a cause-effect graph, is to use abstract syntax trees as a representation of a predicate. The syntax tree is then used to derive tests using one of three procedures described here.

p_r: (printer_status=ON) $\wedge$ (printer_tray=$\neg$ empty)

The predicate p_r consists of two relational expressions joined with the $\wedge$ Boolean operator. Each of the two relational expressions uses the equality (=) symbol. A programmer might code p_r correctly or might make an error thus creating a fault in the program. We are interested in generating test cases from predicates such that any fault, that belongs to a class of faults, is guaranteed to be detected during testing. Testing to ensure that there are no errors in the implementation of predicates is also known as *predicate testing*.

We begin our move towards the test generation algorithm by first defining some basic terms related to predicates and Boolean expressions. Then we will examine the fault model that indicates what faults are the targets of the tests generated by the algorithms presented. This is followed by an introduction to constraints and tests and then the algorithm for which you would have waited so long.

4.4.1 A fault model

Predicate testing, as introduced in this chapter, targets three classes of faults: Boolean operator fault, relational operator fault, and arithmetic

expression fault. A Boolean operator fault is caused when (i) an incorrect Boolean operator is used, (ii) a negation is missing or placed incorrectly, (iii) parentheses are incorrectly placed, and (iv) an incorrect Boolean variable is used. A relational operator fault occurs when an incorrect relational operator is used. An arithmetic expression fault occurs when the value of an arithmetic expression is off by an amount equal to ϵ.

Predicate testing is useful in detecting errors in the use of Boolean and relational operators as well as off-by-1 errors in arithmetic expressions used in a condition.

Given a predicate p_r and a test case t, we write $p(t)$ as an abbreviation for the truth value obtained by evaluating p_r on t. For example, if p_r is $a < b \wedge r > s$ and t is $< a = 1, b = 2, r = 0, s = 4 >$, then $p(t) = \texttt{false}$. Let us now examine a few examples of the faults in the fault model used in this section.

Boolean operator fault: Suppose that the specification of a software module requires action to be performed when the condition $(a < b) \vee (c > d) \wedge e$ is true. Here a, b, c, and d are integer variables and e a Boolean variable. Three incorrect codings of this condition, each containing a Boolean operator fault, are given below.

The incorrect use of a Boolean operator is known as a *Boolean operator fault*.

$(a < b) \wedge (c > d) \wedge e$	Incorrect Boolean operator
$(a < b) \vee \neg(c > d) \wedge e$	Incorrect negation operator
$(a < b) \wedge (c > d) \vee e$	Incorrect Boolean operators
$(a < b) \vee (c > d) \wedge f$	Incorrect Boolean variable (f instead of e).

Notice that a predicate might contain a single or multiple faults. The third example above is a predicate containing two faults.

Relational operator fault: Examples of relational operator faults follow.

The incorrect use of a relational operator is known as a *relational operator fault*.

$(a == b) \vee (c > d) \wedge e$	Incorrect relational operator; $<$ replaced by $==$.
$(a == b) \vee (c \leq d) \wedge e$	Two relational operator faults.
$(a == b) \vee (c > d) \vee e$	Incorrect relational and Boolean operators.

Arithmetic expression fault: We consider three types of off-by-ϵ fault in an arithmetic expression. These are referred to as off-by-ϵ, off-by-ϵ^*, and off-by-ϵ^+. To understand the differences between these three faults, consider a correct relational expression E_c to be e_1 $relop_1$ e_2 and an incorrect relational expression E_i to be $e_3 relop_2$ e_4. We assume that the arithmetic expressions e_1, e_2, e_3, and e_4 contain the same set of variables. The three fault types are defined below.

The arithmetic expression fault models off-by-1 or, in general, off-by "some small amount" errors in arithmetic expressions that are used in a condition.

- E_i has an off-by-ϵ fault if $|e_3 - e_4| = \epsilon$ for any test case for which $e_1 = e_2$.
- E_i has an off-by-ϵ^* fault if $|e_3 - e_4| \geq \epsilon$ for any test case for which $e_1 = e_2$.
- E_i has an off-by-ϵ^+ fault if $|e_3 - e_4| > \epsilon$ for any test case for which $e_1 = e_2$.

Suppose that the correct predicate E_c is $a < b + c$, where a and b are integer variables. Assuming $\epsilon = 1$, three incorrect versions of E_c follow.

$a < b$	Assuming that $c = 1$, there is an off-by-1 fault in E_i as $\lvert a - b \rvert = 1$ for any value of a and b that makes $a = b + c$.
$a < b + 1$	Assuming that $c \geq 2$, there is an off-by-1* fault in E_i as $\lvert a - (b + 1) \rvert \geq 1$ for any value of a and b that makes $a = b + c$.
$a < b - 1$	Assuming that $c > 0$, there is an off-by-1$^+$ fault in E_i as $\lvert a - (b - 1) \rvert > 1$ for any value of a and b that makes $a = b + c$.

Given a correct predicate p_c, the goal of predicate testing is to generate a test set T such that there is at least one test case $t \in T$ for which p_c and its faulty version p_i, evaluate to different truth values. Such a test set is said to guarantee the detection of any fault of the kind in the fault model introduced above.

As an example, suppose that $p_c : a < b + c$ and $p_i : a > b + c$. Consider a test set $T = \{t_1, t_2\}$ where $t_1 :< a = 0, b = 0, c = 0 >$ and $t_2 :< a = 0, b = 1, c = 1 >$. The fault in p_i is not revealed by t_1 as both p_c and p_i evaluate to `false` when evaluated against t_1. However, the fault is revealed by the t_2 as p_c evaluates to `true` and p_i to `false` when evaluated against t_2.

4.4.2 Missing or extra Boolean variable faults

A missing or extra Boolean variable leads to an incorrect path condition.

Two additional types of common faults have not been considered in the fault model described above. These are the missing Boolean variable and the extra Boolean variable faults.

As an illustration, consider a process control system in which the pressure P and temperature T of a liquid container is being measured and transmitted to a control computer. The emergency check in the control software is required to raise an alarm when any one of the following conditions is true: $T > T_{max}$ and $P > P_{max}$. The alarm specification can be translated to a predicate $p_r : T > T_{max} \vee P > P_{max}$ which when true must cause the computer to raise the alarm, and not otherwise.

Notice that p_r can written as a Boolean expression $a + b$ where $a = T > T_{max}$ and $b = P > P_{max}$. Now suppose that the implementation of the control software codes p_r as a and not as $a \vee b$. Obviously, there is a fault in coding p_r. We refer to this fault as the *missing Boolean variable* fault.

Next, assume that the predicate p_r has been incorrectly coded as $a + b + c$ where c is a Boolean variable representing some condition. Once again we have a fault in the coding of p_r. We refer to this fault as *extra Boolean variable* fault.

The missing and extra Boolean variable faults are not guaranteed to be detected by tests generated using any of the procedures introduced in this chapter.

4.4.3 Predicate constraints

Let BR denote the following set of symbols $\{\mathbf{t}, \mathbf{f}, <, =, >, +\epsilon, -\epsilon\}$. Here "BR" is an abbreviation for "Boolean and Relational." We shall refer to an element of the BR set as a BR-symbol.

A BR symbol specifies a constraint on a simple condition. For example, the symbol "f" can be used as a constraint for a < b. To satisfy this constraint, values of a and b must be chosen such that the condition is false.

A BR symbol specifies a *constraint* on a Boolean variable or a relational expression. For example, the symbol "$+\epsilon$" is a constraint on the expression $E' : e_1 < e_2$. Satisfaction of this constraint requires that a test case for E' ensure that $0 < e_1 - e_2 \leq \epsilon$. Similarly, the symbol "$-\epsilon$" is another constraint on E'. This constraint can be satisfied by a test for E' such that $-\epsilon \leq e_1 - e_2 < 0$.

A constraint C is considered *infeasible* for predicate p_r if there exist no input values for the variables in p_r that satisfy C. For example, constraint $(>, >)$ for predicate $a > b \wedge b > d$ requires the simple predicates $a > b$ and $d > a$ to be true. However, this constraint is infeasible if it is known that $d > a$.

A simple condition is constrained to a specific truth value by a predicate constraint. Thus, a compound condition has as many predicate constraints as there are simple condition in it.

Example 4.8 Consider the relational expression $E : a < c + d$. Now consider constraint "$C : (=)$" on E. While testing E for correctness, satisfying C requires at least one test case such that $a = c + d$. Thus the test case $< a = 1, c = 0, d = 1 >$ satisfies the constraint C on E.

As another example, consider the constraint $C : (+\epsilon)$ on expression E given above. Let $\epsilon = 1$. A test to satisfy C requires that $0 < a - (c + d) \leq 1$. Thus, the test case $< a = 4, c = 2, d = 1 >$ satisfies constraint $(+\epsilon)$ on expression E.

Similarly, given a Boolean expression $E : b$, the constraint "**t**" is satisfied by a test case that sets variable b to `true`.

A predicate constraint for a condition is considered infeasible if it cannot be satisfied by any value of the variables in the condition.

BR symbols **t** and **f** are used to specify constraints on Boolean variables and expressions. A constraint on a relational expression is specified using any of the three symbols $<$, $=$, and $>$. Symbols **t** and **f** can also be used to specify constraints on a simple relational expression when the expression is treated as a representative of a Boolean variable. For example, expression $p_r : a < b$ could serve as a representative of a Boolean variable z in which case we can specify constraints on p_r using **t** and **f**.

We will now define constraints for entire predicates that contain Boolean variables and relational expressions joined by one or more Boolean operators.

A predicate constraint for a compound predicate having n simple conditions is a sequence of $(n+1)$ BR symbols, one for each simple condition.

Let p_r denote a predicate with $n, n > 0$, $\wedge$ and $\vee$ operators. A *predicate constraint* C for predicate p_r is a sequence of $(n+1)$ BR symbols, one for each Boolean variable or relational expression in p_r. When clear from context, we refer to "predicate constraint" as simply *constraint*.

We say that a test case t *satisfies* C for predicate p_r, if each component of p_r satisfies the corresponding constraint in C when evaluated against t. Constraint C for predicate p_r guides the development of a test for p_r, i.e. it offers hints on the selection of values of the variables in p_r.

Example 4.9 Consider the predicate $p_r : b \wedge r < s \vee u \geq v$. One possible BR-constraint for p_r is $C : (t, =, >)$. Note that C contains three constraints one for each of the three components of p_r. Constraint t applies to b, $=$ to $r < s$, and $>$ to $u \geq v$. The following test case satisfies C for p_r.

$< b = \texttt{true}, r = 1, s = 1, u = 1, v = 0 >$

Several other test cases also satisfy C for p_r. The following test case does not satisfy C for p_r.

$< b = \texttt{true}, r = 1, s = 1, u = 1, v = 2 >$

as the last of the three elements in C is not satisfied for the corresponding component of p_r which is $u \geq v$.

A test case that satisfies a true constraint makes the corresponding predicate evaluate as true. Similarly, a test case that satisfies a false constraint makes the corresponding predicate evaluate as false.

Given a constraint C for predicate p_r, any test case that satisfies C makes p_r either `true` or `false`. Thus, we write $p_r(C)$ to indicate the value of p_r obtained by evaluating it against any test case that satisfies C. A constraint C for which $p_r(C) = \texttt{true}$ is referred to as a *true constraint* and the one for which $p_r(C)$ evaluates to `false` is referred to as a *false constraint*. We partition a set of constraints S into two sets S^t and S^f such that $S = S^t \cup S^f$. The partition is such that for each $C \in S^t$, $p_r(C) = \texttt{true}$ and for each $C \in S^f$, $p_r(C) = \texttt{false}$.

Example 4.10 Consider the predicate $p_r : (a < b) \wedge (c > d)$ and constraint $C_1 : (=, >)$ on p_r for $\epsilon = 1$. Any test case that satisfies C_1 on p_r, makes p_r evaluate to `false`. Hence C_1 is a false constraint. Consider another constraint $C_2 : (<, +\epsilon)$ for $\epsilon = 1$ on predicate p_r. Any test case that satisfies C_2 on p_r, makes p_r evaluate to `true`. Hence C_2 is a true constraint. Now, if $S = \{C_1, C_2\}$ is a set of constraints on predicate p_r, then we have $S^t = \{C_2\}$ and $S^f = \{C_1\}$.

4.4.4 Predicate testing criteria

We are interested in generating a test set T from a given predicate p_r such that (a) T is minimal and (b) T guarantees the detection of

any fault in the implementation of p_r that conforms to the fault model described earlier. Towards the goal of generating such a test set, we define three criteria commonly known as the BOR, BRO, and BRE testing criteria. The names BOR, BRO, and BRE correspond to, respectively, "**B**oolean **O**perato**r**," "**B**oolean and **R**elational **O**perator," and "**B**oolean and **R**elational **E**xpression." Formal definitions of the three criteria follow.

A BOR adequate test set is guaranteed to detect all errors that correspond to our fault model. Tests that are BRO and BRE adequate are likely to detect errors that correspond to our fault model.

- *A test set T that satisfies the BOR testing criterion for a compound predicate p_r, guarantees the detection of single or multiple Boolean operator faults in the implementation of p_r. T is referred to as a BOR-adequate test set and sometimes written as T_{BOR}.*
- *A test set T that satisfies the BRO testing criterion for a compound predicate p_r, is likely to detect single Boolean operator and relational operator faults in the implementation of p_r. T is referred to as a BRO-adequate test set and sometimes written as T_{BRO}.*
- *test set T that satisfies the BRE testing criterion for a compound predicate p_r, is likely to detect single Boolean operator, relational expression, and arithmetic expression faults in the implementation of p_r. T is referred to as a BRE-adequate test set and sometimes written as T_{BRE}.*

The term "guarantees the detection of" is to be interpreted carefully. Let $T_x, x \in \{BOR, BRO, BRE\}$, be a test set derived from predicate p_r. Let p_f be another predicate obtained from p_r by injecting single or multiple faults of one of three kinds: Boolean operator fault, relational operator fault, and arithmetic expression fault. T_x is said to guarantee the detection of faults in p_f if for some $t \in T_x$, $p(t) \neq p_f(t)$. The next example shows a sample BOR-adequate test set and its fault detection effectiveness.

Example 4.11 Consider the compound predicate $p_r : a < b \wedge c > d$. Let S denote a set of constraints on p_r; $S = \{(\mathbf{t}, \mathbf{t}), (\mathbf{t}, \mathbf{f}), (\mathbf{f}, \mathbf{t})\}$. The following test set T satisfies constraint set S and the BOR testing criterion.

$$
\begin{aligned}
T = \{ & t_1 :< a = 1, b = 2, c = 1, d = 0 >; \text{Satisfies } (\mathbf{t}, \mathbf{t}), \\
& t_2 :< a = 1, b = 2, c = 1, d = 2 >; \text{Satisfies } (\mathbf{t}, \mathbf{f}), \\
& t_3 :< a = 1, b = 0, c = 1, d = 0 >; \text{Satisfies } (\mathbf{f}, \mathbf{t}) \\
\}
\end{aligned}
$$

As T satisfies the BOR testing criterion, it guarantees that all single and multiple Boolean operator faults in p_r will be revealed. Let us

Table 4.2 Fault detection ability of a BOR-adequate test set T of Example 4.11 for single and multiple Boolean operator faults. Results of evaluation that distinguish the faulty predicate from the correct one are underlined.

	Predicate	t_1	t_2	t_3
	$a < b \wedge c > d$	true	false	false
Single Boolean operator fault.				
1	$a < b \vee c > d$	true	<u>true</u>	<u>true</u>
2	$a < b \wedge \neg c > d$	<u>false</u>	<u>true</u>	false
3	$\neg a < b \wedge c > d$	<u>false</u>	false	<u>true</u>
Multiple Boolean operator fault.				
4	$a < b \vee \neg c > d$	true	<u>true</u>	false
5	$\neg a < b \vee c > d$	true	false	<u>true</u>
6	$\neg a < b \wedge \neg c > d$	<u>false</u>	false	false
7	$\neg a < b \vee \neg c > d$	false	<u>true</u>	<u>true</u>

check this by evaluating p_r, and all its variants created by inserting Boolean operator faults, against T.

Table 4.2 lists p_r and a total of 7 faulty predicates obtained by inserting single and multiple Boolean operator faults in p_r. Each predicate is evaluated against the three test cases in T. Note that each faulty predicate evaluates to a value different from that of p_r for at least one test case.

It is easily verified that if any column is removed from Table 4.2, at least one of the faulty predicates will be left indistinguishable by the tests in the remaining two columns. For example, if we remove test t_2 then the faulty predicate 6 cannot be distinguished from p_r by tests t_1 and t_3. We have thus shown that T is minimal and BOR adequate for predicate p_r.

Exercise 4.11 is similar to the one above and asks you to verify that the two given test sets are BRO and BRE adequate, respectively. In the next section we provide algorithms for generating BOR, BRO, and BRE adequate tests.

4.4.5 BOR, BRO, and BRE adequate tests

We are now ready to describe the algorithms that generate constraints to test a predicate. The actual test cases are generated using the constraints. Recall that a feasible constraint can be satisfied by one or more test cases. Thus, the algorithms we describe are to generate constraints and do not focus on the generation of the specific test cases. The test cases to satisfy

the constraints can be generated manually or automatically. Let us begin by describing how to generate a BOR constraint set for a give predicate.

Consider the following two definitions of set products. The product of two finite sets A and B, denoted as $A \times B$ is defined as follows.

$$A \times B = \{(a, b) \mid a \in A, b \in B\} \tag{4.1}$$

We need another set product in order to be able to generate minimal sets of constraints. The *onto* set product operator, written as $\otimes$, is defined as follows. For finite sets A and B, $A \otimes B$ is a minimal set of pairs (u, v) such that $u \in A$, $v \in B$, and each element of A appears at least once as u and each element of B appears at least once as v. Note that there are several ways to compute $A \otimes B$ when both A and B contain two or more elements.

The cross product of two sets is unique. However, the onto product of two sets may not be unique.

Example 4.12 Let $A = \{\mathbf{t}, =, >\}$ and $B = \{\mathbf{f}, <\}$. Using the definitions of set product and the onto product, we get the following sets:

$$A \times B = \{(\mathbf{t}, \mathbf{f}), (\mathbf{t}, <), (=, \mathbf{f}), (=, <), (>, \mathbf{f}), (>, <)\}$$

$$A \otimes B = \{(\mathbf{t}, \mathbf{f}), (=, <), (>, <)\}; \text{One possibility.}$$

$$A \otimes B = \{(\mathbf{t}, <), (=, \mathbf{f}), (>, <)\}; \text{Second possibility.}$$

$$A \otimes B = \{(\mathbf{t}, \mathbf{f}), (=, <), (>, \mathbf{f})\}; \text{Third possibility.}$$

$$A \otimes B = \{(\mathbf{t}, <), (=, <), (>, \mathbf{f})\}; \text{Fourth possibility.}$$

$$A \otimes B = \{(\mathbf{t}, <), (=, \mathbf{f}), (>, \mathbf{f})\}; \text{Fifth possibility.}$$

$$A \otimes B = \{(\mathbf{t}, \mathbf{f}), (=, \mathbf{f}), (>, <)\}; \text{Sixth possibility.}$$

Notice that there are six different ways to compute $A \otimes B$. The algorithms described next select any one of the different sets.

Given a predicate p_r, the generation of the BOR, BRO, and BRE constraint sets requires the abstract syntax tree for p_r denoted as $AST(p_r)$. Recall that (a) each leaf node of $AST(p_r)$ represents a Boolean variable or a relational expression and (b) an internal node of $AST(p_r)$ is a Boolean operator such as $\wedge$, $\vee$, , and $\neg$ known as an *AND* node, *OR* node, *XOR* node, and *NOT* node, respectively.

We now introduce four procedures for generating tests from a predicate. The first three procedures generate BOR, BRO, and BRE adequate tests for predicates that involve only singular expressions. The last procedure, named BOR-MI, generates tests for predicates that contain at least one non-singular expression. See Exercise 4.20 for an example that illustrates

the problem in applying the first three procedures below to a non-singular expression.

The BOR constraint set

An abstract syntax tree (AST) captures the syntactic relationships among the constituent elements of a predicate. The well-known parsing technique, often used in compilers, can be used to construct an AST for any predicate.

Let p_r be a predicate and $AST(p_r)$ its abstract syntax tree. We use letters such as N, N_1, and N_2 to refer to various nodes in the $AST(p_r)$. S_N denotes the constraint set attached to node N. As explained earlier, S_N^t and S_N^f denote, respectively, the true and false constraint sets associated with node N; $S_N = S_N^t \cup S_N^f$. The following algorithm generates the BOR constraint set for p_r.

A BOR constraint set consists of n-tuples of BR symbols t and f, where n is the number of simple conditions in the predicate.

The BOR constraint set is generated by assigning constraints to the leaf nodes of an abstract syntax tree. Constraints at the leaves are then propagated up the tree using rules defined in the algorithm. The derived constraint set as available at the root node.

`Procedure for generating a minimal BOR constraint set from an abstract syntax tree of a predicate` p_r.

Input: An abstract syntax tree for predicate p_r, denoted by $AST(p_r)$. p_r, contains only singular expressions.

Output: BOR constraint set for p_r attached to the root node of $AST(p_r)$.

Procedure: `BOR-CSET`

Step 1 Label each leaf node N of $AST(p_r)$ with its constraint set $S(N)$ for each leaf $S_N = \{\mathbf{t}, \mathbf{f}\}$.

Step 2 Visit each non-leaf node in $AST(p_r)$ in a bottom up manner. Let N_1 and N_2 denote the direct descendants of node N, if N is an AND- or an OR-node. If N is a NOT-node, then N_1 is its direct descendant. S_{N_1} and S_{N_2} are the BOR constraint sets for nodes N_1 and N_2, respectively. For each non-leaf node N, compute S_N as follows:

2.1 *N is an OR-node:*
$S_N^f = S_{N_1}^f \otimes S_{N_2}^f$
$S_N^t = (S_{N_1}^t \times \{f_2\}) \cup (\{f_1\} \times S_{N_2}^t)$
where $f_1 \in S_{N_1}^f$ and $f_2 \in S_{N_2}^f$

2.2 *N is an AND-node:*
$S_N^t = S_{N_1}^t \otimes S_{N_2}^t$
$S_N^f = (S_{N_1}^f \times \{t_2\}) \cup (\{t_1\} \times S_{N_2}^f)$
where $t_1 \in S_{N_1}^t$ and $t_2 \in S_{N_2}^t$

2.3 *N is NOT-node:*
$S_N^t = S_{N_1}^f$
$S_N^f = S_{N_1}^t$

Step 3 The constraint set for the root of $AST(p_r)$ is the desired BOR constraint set for p_r.

`End of Procedure BOR-CSET`

Figure 4.11 BOR constraint sets for predicates (a) $a < b \wedge c > d$ and (b) $(a + b) < c \wedge \neg p \vee (r > s)$. The constraint sets are listed next to each node. See the text for the separation of each constraint set into its true and false components.

Example 4.13 Let us apply the procedure described above to generate the BOR-constraint sets for the predicate $p_1 : a < b \wedge c > d$ used in Example 4.11. The abstract syntax tree for p_1 is shown in Figure 4.11(a).

N_1 and N_2 are the leaf nodes. The constraint sets for these two nodes are given below.

$$S^t_{N_1} = \{\mathbf{t}\}, \qquad S^f_{N_1} = \{\mathbf{f}\}$$
$$S^t_{N_2} = \{\mathbf{t}\}, \qquad S^f_{N_2} = \{\mathbf{f}\}$$

Traversing $AST(p_1)$ bottom up, we compute the constraint set for non-leaf N_3 which is an AND-node.

$$\begin{aligned} S^t_{N_3} &= S^t_{N_1} \otimes S^t_{N_2} \\ &= \{\mathbf{t}\} \otimes \{\mathbf{t}\} \\ &= \{(\mathbf{t}, \mathbf{t})\} \\ S^f_{N_3} &= (S^f_{N_1} \times \{t_2\}) \cup (\{t_1\} \times S^f_{N_2}) \\ &= (\{\mathbf{f}\} \times \{\mathbf{t}\}) \cup (\{\mathbf{t}\} \times \{\mathbf{f}\}) \\ &= \{(\mathbf{f}, \mathbf{t}), (\mathbf{t}, \mathbf{f})\} \end{aligned}$$

Thus, we obtain $S_{N_3} = \{(\mathbf{t},\mathbf{t}), (\mathbf{f},\mathbf{t}), (\mathbf{t},\mathbf{f})\}$ which is the BOR-constraint set for predicate p_1. We have now shown how S_{N_3}, used in Example 4.11, is derived using a formal procedure.

An AST is traversed bottom-up to derive the BOR constraint set for a predicate. Given that the onto set product is used in the construction of the constraint set, the resulting set may not be unique.

Example 4.14 Let us compute the BOR constraint set for predicate $p_2 : (a + b < c) \wedge \neg p \vee (r > s)$ which is a bit more complex than predicate p_1 from the previous example. Note that the $\wedge$ operator takes priority over the $\vee$ operator. Hence p_2 is equivalent to the expression $((a + b < c) \wedge (\neg p)) \vee (r > s)$.

First, as shown in Figure 4.11(b), we assign the default BOR constraint sets to the leaf nodes N_1, N_2, and N_5. Next we traverse $AST(p_2)$ bottom up and breadth first. Applying the rule for a NOT node, we obtain the BOR constraint set for N_3 as follows:

$$S^t_{N_3} = S^f_{N_2} = \{\mathbf{f}\}$$
$$S^f_{N_3} = S^t_{N_2} = \{\mathbf{t}\}$$

The following BOR constraint set is obtained by applying the rule for the AND-node.

$$\begin{aligned}
S^t_{N_4} &= S^t_{N_1} \otimes S^t_{N_3} \\
&= \{\mathbf{t}\} \otimes \{\mathbf{f}\} \\
&= \{(\mathbf{t},\mathbf{f})\} \\
S^f_{N_4} &= (S^f_{N_1} \times \{t_{N_3}\}) \cup (\{t_{N_1}\} \times S^f_{N_3}) \\
&= (\{\mathbf{f}\} \times \{\mathbf{f}\}) \cup (\{\mathbf{t}\} \times \{\mathbf{t}\}) \\
&= \{(\mathbf{f},\mathbf{f}), (\mathbf{t},\mathbf{t})\} \\
S_{N_4} &= \{(\mathbf{t},\mathbf{f}), (\mathbf{f},\mathbf{f}), (\mathbf{t},\mathbf{t})\}
\end{aligned}$$

Using the BOR constraint sets for nodes N_4 and N_5 and applying the rule for OR-node, we obtain the BOR constraint set for node N_6 as follows:

$$\begin{aligned}
S^f_{N_6} &= S^f_{N_4} \otimes S^f_{N_5} \\
&= \{(\mathbf{f},\mathbf{f}), (\mathbf{t},\mathbf{t})\} \otimes \{\mathbf{f}\} \\
&= \{(\mathbf{f},\mathbf{f},\mathbf{f}), (\mathbf{t},\mathbf{t},\mathbf{f})\} \\
S^t_{N_6} &= (S^t_{N_4} \times \{f_{N_5}\}) \cup (\{f_{N_4}\} \times S^t_{N_5}) \\
&= (\{(\mathbf{t},\mathbf{f})\} \times \{\mathbf{f}\}) \cup (\{(\mathbf{f},\mathbf{f})\} \times \{\mathbf{t}\}) \\
&= \{(\mathbf{t},\mathbf{f},\mathbf{f}), (\mathbf{f},\mathbf{f},\mathbf{t})\} \\
S_{N_6} &= \{(\mathbf{t},\mathbf{f},\mathbf{f}), (\mathbf{f},\mathbf{f},\mathbf{t}), (\mathbf{t},\mathbf{t},\mathbf{f}), (\mathbf{f},\mathbf{f},\mathbf{f})\}
\end{aligned}$$

Table 4.3 Sample tests cases that satisfy the BOR constraints for predicate p_2 derived in Example 4.14.

	$a+b<c$	p	$r>s$	Test case
t_1	**t**	**f**	**f**	$<a=1, b=1, c=3, p=$ `false`, $r=1, s=2>$
t_2	**f**	**f**	**t**	$<a=1, b=1, c=1, p=$ `false`, $r=1, s=0>$
t_3	**t**	**t**	**f**	$<a=1, b=1, c=3, p=$ `true`, $r=1, s=1>$
t_4	**f**	**f**	**f**	$<a=1, b=1, c=0, p=$ `false`, $r=0, s=0>$

A test set for a predicate is derived from its BOR constraint set by selecting suitable values of the variables involved in the predicate.

Notice that we could select any one of the constraints (**f**, **f**) or (**t**, **t**) for f_{N_4}. Here we have arbitrarily selected (**f**, **f**). Sample tests for p_2 that satisfy the four BOR constraints are given in Table 4.3. Exercise 4.13 asks you to confirm that indeed the test set in Table 4.3 is adequate with respect to the BOR testing criterion.

The BRO constraint set

Recall that a test set adequate with respect to a BRO constraint set for predicate, p_r, is very likely to detect all combinations of single or multiple Boolean operator and relational operator faults. The BRO constraint set S for a relational expression e_1 *relop* e_2 is $\{(>), (=), (<)\}$. As shown below, the separation of S into its true and false components depends on *relop*.

The BRO constraint set is useful when deriving tests for predicates that involve relational operators.

$$
\begin{array}{lll}
relop:> & S^t=\{(>)\} & S^f=\{(=),(<)\} \\
relop:\geq & S^t=\{(>),(=)\} & S^f=\{(<)\} \\
relop:= & S^t=\{(=)\} & S^f=\{(<),(>)\} \\
relop:< & S^t=\{(<)\} & S^f=\{(=),(>)\} \\
relop:\leq & S^t=\{(<),(=)\} & S^f=\{(>)\}
\end{array}
$$

We now modify Procedure BOR-CSET introduced earlier for the generation of the minimal BOR constraint set to generate a minimal BRO constraint set. The modified procedure follows.

`Procedure for generating a minimal BRO constraint set from an abstract syntax tree of a predicate` p_r.

Input: An abstract syntax tree for predicate p_r, denoted by $AST(p_r)$. p_r, contains only singular expressions.

The algorithm for deriving a BRO constraint set is similar to that for deriving a BOR constraint set. The primary difference is in the constraints assigned to the leaf nodes.

Output: BRO constraint set for p_r attached to the root node of $AST(p_r)$.

Procedure: `BRO-CSET`

Step 1 Label each leaf node N of $AST(p_r)$ with its constraint set S_N. For each leaf node that represents a Boolean variable, $S_N = \{\mathbf{t}, \mathbf{f}\}$. For each leaf node that is a relational expression, $S_N = \{(>), (=), (<)\}$.

Step 2 Visit each non-leaf node in $AST(p_r)$ in a bottom up manner. Let N_1 and N_2 denote the direct descendants of node N, if N is an AND- or an OR-node. If N is a NOT-node, then N_1 is its direct descendant. S_{N_1} and S_{N_2} are the BRO constraint sets for nodes N_1 and N_2, respectively. For each non-leaf node N, compute S_N as per Steps 2.1, 2.2, and 2.3 in Procedure BRO-CSET.

Step 3 The constraint set for the root of $AST(p_r)$ is the desired BRO constraint set for p_r.

`End of Procedure BRO-CSET`

Example 4.15 Let us construct a BRO constraint set for predicate $p_r : (a + b) < c \wedge \neg p \vee (r > s)$. Figure 4.12 shows the abstract syntax tree for p_r with nodes labeled by the corresponding BRO constraint sets. Let us derive these sets using Procedure BRO-CSET.

First, the leaf nodes are labeled with the corresponding BRO constraint sets depending on their type. Next we traverse the tree bottom up and derive the BRO constraints for each node from those of its immediate descendants. The BRO constraint set for node N_3,

Figure 4.12 BRO constraint set for predicate $p_r = (a + b) < c \wedge \neg p \vee (r > s)$. The constraint sets are listed next to each node. See the text for the separation of each constraint set into its true and false components.

as shown in Figure 4.12, is derived using Step 2.3 of Procedure BOR-CSET.

Next we construct the BRO constraint set for the AND node N_4 using Step 2.2.

$$
\begin{aligned}
S^t_{N_4} &= S^t_{N_1} \otimes S^t_{N_3} \\
&= \{(<)\} \otimes \{\mathbf{f}\} \\
&= \{(<, \mathbf{f})\} \\
S^f_{N_4} &= (S^f_{N_1} \times \{t_{N_3}\}) \cup (\{t_{N_1}\} \times S^f_{N_3}) \\
&= (\{(>), (=)\} \times \{\mathbf{f}\}) \cup (\{(<)\} \times \{\mathbf{t}\}) \\
&= \{(>, \mathbf{f}), (=, \mathbf{f}), (<, \mathbf{t})\} \\
S_{N_4} &= \{(<, \mathbf{f}), (>, \mathbf{f}), (=, \mathbf{f}), (<, \mathbf{t})\}
\end{aligned}
$$

Finally we construct the BRO constraint set for the root node N_6 by applying Step 2.1 to the BRO constraint sets of nodes N_4 and N_5.

$$
\begin{aligned}
S^f_{N_6} &= S^f_{N_4} \otimes S^f_{N_5} \\
&= \{(>, \mathbf{f}), (=, \mathbf{f}), (<, \mathbf{t})\} \otimes \{(=), (<)\} \\
&= \{(>, \mathbf{f}, =), (=, \mathbf{f}, <), (<, \mathbf{t}, =)\} \\
S^t_{N_6} &= (S^t_{N_4} \times \{f_{N_5}\}) \cup (\{f_{N_4}\} \times S^t_{N_5}) \\
&= (\{(<, \mathbf{f})\} \times \{(=)\}) \cup (\{(>, \mathbf{f})\} \times \{(>)\}) \\
&= \{(<, \mathbf{f}, =), (>, \mathbf{f}, >)\} \\
S_{N_6} &= \{(>, \mathbf{f}, =), (=, \mathbf{f}, <), (<, \mathbf{t}, =), (<, \mathbf{f}, =), (>, \mathbf{f}, >)\}
\end{aligned}
$$

Sample tests for p_r that satisfy the five BRO constraints are given in Table 4.4. Exercise 4.14 asks you to confirm that indeed the test set in Table 4.4 is adequate with respect to the BRO testing criterion.

The BRE constraint set

We now show how to generate BRE constraints that lead to test cases which are likely to detect any Boolean operator, relation operator, arithmetic expression, or a combination thereof, faults in a predicate. The BRE constraint set for a Boolean variable remains $\{\mathbf{t}, \mathbf{f}\}$ as with the BOR and BRO constraint sets. The BRE constraint set for a relational expression is $\{(+\epsilon), (=), (-\epsilon)\}, \epsilon > 0$. The BRE constraint set S for a relational expression e_1 *relop* e_2 is separated into subsets S^t and S^f based on the following relations.

A BRE constraint set is useful when deriving tests for relational expressions that contain arithmetic operator and are to be tested for errors such as "off by 1".

Table 4.4 Sample tests cases that satisfy the BRO constraints for predicate p_r derived in Example 4.15.

	$a+b<c$	p_r	$r>s$	Test case
t_1	$>$	**f**	$=$	$<a=1, b=1, c=1, p_r=$ `false`, $r=1, s=1>$
t_2	$=$	**f**	$<$	$<a=1, b=0, c=1, p_r=$ `false`, $r=1, s=2>$
t_3	$<$	**t**	$=$	$<a=1, b=1, c=3, p_r=$ `true`, $r=1, s=1>$
t_4	$<$	**f**	$=$	$<a=0, b=2, c=3, p_r=$ `false`, $r=0, s=0>$
t_5	$>$	**f**	$>$	$<a=1, b=1, c=0, p_r=$ `false`, $r=2, s=0>$

Constraint	Satisfying condition
$+\epsilon$	$0<e_1-e_2\leq+\epsilon$
$-\epsilon$	$-\epsilon\leq e_1-e_2<0$

Based on the conditions listed above, we can now separate the BRE constraint S into its true and false components as follows:

$$
\begin{array}{lll}
relop:> & S^t=\{(+\epsilon)\} & S^f=\{(=),(-\epsilon)\} \\
relop:\geq & S^t=\{(+\epsilon),(=)\} & S^f=\{(-\epsilon)\} \\
relop:= & S^t=\{(=)\} & S^f=\{(+\epsilon),(-\epsilon)\} \\
relop:< & S^t=\{(-\epsilon)\} & S^f=\{(=),(+\epsilon)\} \\
relop:\leq & S^t=\{(-\epsilon),(=)\} & S^f=\{(+\epsilon)\}
\end{array}
$$

The procedure to generate a minimal BRE-constraint set is similar to Procedure BRO-CSET. The key difference in the two procedures lies in the construction of the constraint sets for the leaves. Procedure BRE-CSET follows.

`Procedure for generating a minimal BRE constraint set from an abstract syntax tree of a predicate` p_r.

Input: An abstract syntax tree for predicate p_r, denoted by $AST(p_r)$. p_r contains only singular expressions.

Output: BRE constraint set for p_r attached to the root node of $AST(p_r)$.

Procedure: `BRE-CSET`

Step 1 Label each leaf node N of $AST(p_r)$ with its constraint set S_N. For each leaf node that represents a Boolean variable, $S_N = \{\mathbf{t}, \mathbf{f}\}$. For each leaf node that is a relational expression, $S_N = \{(+\epsilon), (=), (-\epsilon)\}$.

Step 2 Visit each non-leaf node in $AST(p_r)$ in a bottom up manner. Let N_1 and N_2 denote the direct descendants of node N, if N is an AND- or an OR-node. If N is a NOT-node, then N_1 is its direct descendant. S_{N_1} and S_{N_2} are the BRE constraint sets for nodes N_1 and N_2, respectively. For each non-leaf node N, compute S_N as in Steps 2.1, 2.2, and 2.3 in Procedure BOR-CSET.

Step 3 The constraint set for the root of $AST(p_r)$ is the desired BRE constraint set for p_r.

`End of Procedure` BRE-CSET

Example 4.16 Consider the predicate $p : (a + b) < c \wedge \neg p \vee r > s$. The BRE constraint set for p_r is derived in Figure 4.13. Notice the similarity of BRE and BRO constraints listed against the corresponding nodes in Figures 4.12 and 4.13.

Note that tests t_1 through t_4 in Table 4.5 are identical to the corresponding tests in Table 4.4. However, t_5 in Table 4.4 does not satisfy the $(+\epsilon)$ constraint. Hence, we have a different t_5 in Table 4.5. Exercise 4.17 asks you to compare the test cases derived from constraints using BRO-CSET and BRE-CSET.

4.4.6 BOR constraints for non-singular expressions

As mentioned earlier, the test generation procedures described in the previous sections generate BOR, BRO, and BRE adequate tests for predicates

While constraint sets for non-singular predicates can be constructed directly by applying the procedures given earlier, more effective procedures are available.

Figure 4.13 BRE constraint set for predicate $(a + b) < c \wedge \neg p \vee (r > s)$.

Table 4.5 Sample tests cases that satisfy the BRE constraints for predicate p_r derived in Example 4.16 $\epsilon = 1$.

	$a+b<c$	p	$r>s$	Test case
t_1	$+\epsilon$	**f**	$=$	$< a=1, b=1, c=1, p=$ `false`, $r=1, s=1 >$
t_2	$=$	**f**	$-\epsilon$	$< a=1, b=0, c=1, p=$ `false`, $r=1, s=2 >$
t_3	$-\epsilon$	**t**	$=$	$< a=1, b=1, c=3, p=$ `true`, $r=1, s=1 >$
t_4	$-\epsilon$	**f**	$=$	$< a=0, b=2, c=3, p=$ `false`, $r=0, s=0 >$
t_5	$+\epsilon$	**f**	$+\epsilon$	$< a=1, b=1, c=1, p=$ `false`, $r=2, s=0 >$

A non-singular expression contains multiple occurrences of one or more literals. The BOR-CSET algorithm needs to be modified to generate constraints for such an expression.

that do not contain any non-singular expressions. However, the presence of non-singular expressions might create conflicts while merging constraints during the traversal of the abstract syntax tree (see Exercise 4.20). Resolution of such conflicts leads to a constraint set that is not guaranteed to detect all Boolean operator faults that might arise in the implementation of the predicate under test. In this section, we generalize the BOR-CSET procedure to handle predicates that might contain non-singular expressions.

Recall from Section 2.1 that a non-singular expression is one that contains multiple occurrences of a Boolean variable. For example, we list below a few non-singular expressions and their disjunctive normal forms; note that we have omitted the AND operator, used + to indicate the OR operator, and the over bar symbol to indicate the complement of a literal.

There are standard algorithms for converting expressions from DNF to CNF form and vise-versa. These algorithms are not described in this book.

Predicate (p_r)	DNF	Mutually singular components of p_r
$ab(b+c)$	$abb+abc$	$a; b(b+c)$
$a(bc+bd)$	$abc+abd$	$a; (bc+bd)$
$a(bc+\overline{b}+de)$	$abc+a\overline{b}+ade$	$a; bc+\overline{b}; de$

The modified BOR strategy to generate tests from predicate p_r uses the BOR-CSET procedure and another procedure which we refer to as the Meaning Impact, or simply MI, procedure. Before we illustrate the BOR-MI strategy, we introduce the MI procedure for generating tests from any Boolean expression p_r, singular or non-singular. The predicate p_r must be in its minimal disjunctive normal form. If p_r is not in DNF, then it needs

to be transformed into one prior to the application of the MI-CSET procedure described next; there are standard algorithms for transforming any Boolean expression into an equivalent minimal DNF.

A logical expression, i.e. a predicate, is in disjunctive normal form when its constituents are combined using the OR logical operator. For example, if x, y, and z denote simple conditions, then (x OR y or z) is in disjunctive normal form and can also be written as $(x+y+z)$.

The MI-CSET procedure generates a minimal constraint set for a predicate expressed in minimal disjunctive form. Tests generated from such constraint set guarantee the detection of missing or extra negation (NOT) operator.

`Procedure for generating a minimal constraint set from a predicate possibly containing non-singular expressions.`

Input: A Boolean expression $E = e_1 + e_2 + \ldots e_n$ in minimal disjunctive normal form containing n terms. Term e_i, $1 \le i \le n$ contains $l_i > 0$ literals.

Output: A set of constraints S_E that guarantees the detection of missing or extra NOT operator fault in a faulty version of E.

Procedure: `MI-CSET`

Step 1 For each term e_i, $1 \le i \le n$, construct T_{e_i} as the set of constraints that make e_i true.

Step 2 Let $TS_{e_i} = T_{e_i} - \cup_{j=1, j \ne i}^{n} T_{e_j}$. Note that for $i \ne j$, $TS_{e_i} \cap TS_{e_j} = \emptyset$.

Step 3 Construct S_E^t by including one constraint from each TS_{e_i}, $1 \le i \le n$. Note that for each constraint $c \in S_E^t$, $E(c) =$ `true`.

Step 4 Let e_i^j denotes the term obtained by complementing the jth literal in term e_i, for $1 \le i \le n$ and $1 \le j \le l_i$. We count the literals in a term from left to right, the leftmost literal being the first. Construct $F_{e_i^j}$ as the set of constraints that make e_i^j true.

Step 5 Let $FS_{e_i^j} = F_{e_i^j} - \cup_{k=1}^{n} T_{e_k}$. Thus, for any constraint $c \in FS_{e_i^j}$, $E(c) =$ `false`.

Step 6 Construct S_E^f that is minimal and covers each $FS_{e_i^j}$ at least once.

Step 7 Construct the desired constraint set for E as $S_E = S_E^t \cup S_E^f$.

`End of Procedure MI-CSET`

Example 4.17 Let $E = a(bc + \bar{b}d)$, where a, b, c, and d are Boolean variables. Note that E is non-singular as variable b occurs twice. The DNF equivalent of E is $e_1 + e_2$, where $e_1 = abc$ and $e_2 = a\bar{b}d$. We now apply Procedure MI-CSET to generate sets S_E^t and S_E^f. First, let us construct T_{e_1} and T_{e_2}.

$$T_{e_1} = \{(\mathbf{t}, \mathbf{t}, \mathbf{t}, \mathbf{t}), (\mathbf{t}, \mathbf{t}, \mathbf{t}, \mathbf{f})\}$$

$$T_{e_2} = \{(\mathbf{t}, \mathbf{f}, \mathbf{t}, \mathbf{t}), (\mathbf{t}, \mathbf{f}, \mathbf{f}, \mathbf{t})\}$$

Next we construct TS_{e_1} and TS_{e_2}.

$$TS_{e_1} = \{(\mathbf{t},\mathbf{t},\mathbf{t},\mathbf{t}),(\mathbf{t},\mathbf{t},\mathbf{t},\mathbf{f})\}$$
$$TS_{e_2} = \{(\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{t}),(\mathbf{t},\mathbf{f},\mathbf{f},\mathbf{t})\}$$

Notice that as T_{e_1} and T_{e_2} are disjoint, we get $TS_{e_1} = T_{e_1}$ and $TS_{e_2} = T_{e_2}$. Selecting one test each from TS_{e_1} and TS_{e_2}, we obtain a minimal set of constraints that make E true and cover each term of E as follows:

$$S_E^t = \{(\mathbf{t},\mathbf{t},\mathbf{t},\mathbf{f}),(\mathbf{t},\mathbf{f},\mathbf{f},\mathbf{t})\}$$

Note that there exist four possible S_E^t. Next we construct constraints $F_{e_i^j}, 1 \le i \le 2, 1 \le j \le 3$. We have three terms corresponding to e_1, namely $e_1^1 = \overline{a}bc$, $e_1^2 = a\overline{b}c$, and $e_1^3 = ab\overline{c}$. Similarly, we get three terms for e_2, namely, $e_2^1 = \overline{a}\overline{b}d$, $e_2^2 = abd$, and $e_2^3 = a\overline{b}\overline{d}$.

$$F_{e_1^1} = \{(\mathbf{f},\mathbf{t},\mathbf{t},\mathbf{t}),(\mathbf{f},\mathbf{t},\mathbf{t},\mathbf{f})\}$$
$$F_{e_1^2} = \{(\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{t}),(\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{f})\}$$
$$F_{e_1^3} = \{(\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{t}),(\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{f})\}$$
$$F_{e_2^1} = \{(\mathbf{f},\mathbf{f},\mathbf{t},\mathbf{t}),(\mathbf{f},\mathbf{f},\mathbf{f},\mathbf{t})\}$$
$$F_{e_2^2} = \{(\mathbf{t},\mathbf{t},\mathbf{t},\mathbf{t}),(\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{t})\}$$
$$F_{e_2^3} = \{(\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{f}),(\mathbf{t},\mathbf{f},\mathbf{f},\mathbf{f})\}$$

Next we remove from the above six sets any test cases that belong to any of the sets $T_{e_k}, 1 \le k \le n$ sets.

$$FS_{e_1^1} = F_{e_1^1}$$
$$FS_{e_1^2} = \{(\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{f})\}$$
$$FS_{e_1^3} = F_{e_1^3}$$
$$FS_{e_2^1} = F_{e_2^1}$$
$$FS_{e_2^2} = \{(\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{t})\}$$
$$FS_{e_2^3} = F_{e_2^3}$$

The test set that makes E false is now constructed by selecting tests from the six sets listed above such that S_E^f is minimal and covers each FS.

$$S_E^f = \{(\mathbf{f},\mathbf{t},\mathbf{t},\mathbf{f}),(\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{f})(\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{t}),(\mathbf{f},\mathbf{f},\mathbf{t},\mathbf{t})\}$$

The set of constraints S_E generated by the MI-CSET procedure for expression E contains a total of six constraints as follows:

$$S_E = \{(\mathbf{t},\mathbf{t},\mathbf{t},\mathbf{f}),(\mathbf{t},\mathbf{f},\mathbf{f},\mathbf{t}),(\mathbf{f},\mathbf{t},\mathbf{t},\mathbf{f}),(\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{f}),(\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{t}),(\mathbf{f},\mathbf{f},\mathbf{t},\mathbf{t})\}$$

We are now ready to introduce the BOR-MI-CSET procedure for generating a minimal constraint set from a possibly non-singular expression. The procedure described next uses the BOR-CSET and the MI-CSET procedures described earlier.

The BOR-MI-CST procedure generates constraint set for a possibly non-singular logical expression. Tests generated from such a constraint set guarantee the detection of Boolean operator faults.

`Procedure for generating a minimal constraint set for a predicate possibly containing non-singular expressions.`

Input: A Boolean expression E.

Output: A set of constraints S_E that guarantees the detection of Boolean operator faults in E.

Procedure: `BOR-MI-CSET`

Step 1 Partition E into a set of n mutually singular components $E = \{E_1, E_2, \ldots E_n\}$.

Step 2 Generate the BOR constraint set for each singular component in E using the BOR-CSET procedure.

Step 3 Generate the MI-constraint set for each non-singular component in E using the BOR-CSET procedure.

Step 4 Combine the constraints generated in the previous two steps using Step 2 from the BOR-CSET procedure to obtain the constraint set for E.

`End of Procedure BOR-MI-CSET`

The following example illustrates the BOR-MI-CSET procedure.

Example 4.18 As in Example 4.17, let $E = a(bc + \overline{b}d)$, where a, b, c, and d are Boolean variables. Note that E is non-singular as variable b occurs twice. We follow the BOR-MI-CSET procedure to generate the constraint set S_E.

From Step 1, partition E into components e_1 and e_2, where $e_1 = a$ is a singular expression and $e_2 = (bc + \overline{b}d)$ is a non-singular expression. In accordance with Step 2, use the BOR-CSET procedure to find S_{e_1} as follows:

$$S^t_{e_1} = \{(\mathbf{t})\}; S^f_{e_1} = \{(\mathbf{f})\}$$

Next apply the MI-CSET procedure to generate the MI constraints for e_2. Note that e_2 is a DNF expression and can be written as $e_2 = u + v$ where $u = bc$ and $v = \overline{b}d$. Applying Step 1 we obtain the following:

$$T_u = \{(\mathbf{t}, \mathbf{t}, \mathbf{t}), (\mathbf{t}, \mathbf{t}, \mathbf{f})\}$$
$$T_v = \{(\mathbf{f}, \mathbf{t}, \mathbf{t}), (\mathbf{f}, \mathbf{f}, \mathbf{t})\}$$

Applying Step 2 from the MI-CSET procedure to T_u and T_v, and then Step 3, we obtain the following sets:

$$TS_u = T_u$$
$$TS_v = T_v$$
$$S^t_{e_2} = \{(\mathbf{t}, \mathbf{t}, \mathbf{f}), (\mathbf{f}, \mathbf{t}, \mathbf{t})\}$$

Notice that there are several alternatives for $S^t_{e_2}$ from which we need to select one. Next, derive the false constraint set $S^f_{e_2}$ by applying Steps 4, 5, and 6. The complemented subexpressions needed in these steps are $u^1 = \overline{b}c$, $u^2 = b\overline{c}$, $v^1 = bd$, and $v^2 = \overline{b}\,\overline{d}$.

$$F_{u^1} = \{(\mathbf{f}, \mathbf{t}, \mathbf{t}), (\mathbf{f}, \mathbf{t}, \mathbf{f})\}$$
$$F_{u^2} = \{(\mathbf{t}, \mathbf{f}, \mathbf{t}), (\mathbf{t}, \mathbf{f}, \mathbf{f})\}$$
$$F_{v^1} = \{(\mathbf{t}, \mathbf{t}, \mathbf{t}), (\mathbf{t}, \mathbf{f}, \mathbf{t})\}$$
$$F_{v^2} = \{(\mathbf{f}, \mathbf{t}, \mathbf{f}), (\mathbf{f}, \mathbf{f}, \mathbf{f})\}$$
$$FS_{u^1} = \{(\mathbf{f}, \mathbf{t}, \mathbf{f})\}$$
$$FS_{u^2} = \{(\mathbf{t}, \mathbf{f}, \mathbf{t}), (\mathbf{t}, \mathbf{f}, \mathbf{f})\}$$
$$FS_{v^1} = \{(\mathbf{t}, \mathbf{f}, \mathbf{t})\}$$
$$FS_{v^2} = \{(\mathbf{f}, \mathbf{t}, \mathbf{f}), (\mathbf{f}, \mathbf{f}, \mathbf{f})\}$$
$$S^f_{e_2} = \{(\mathbf{f}, \mathbf{t}, \mathbf{f}), (\mathbf{t}, \mathbf{f}, \mathbf{t})\}$$

Here is a summary of what we have derived so far by applying Steps 1, 2, and 3 from procedure BOR-MI-CSET.

$S^t_{e_1} = \{\mathbf{t}\}$ From procedure BOR-CSET.
$S^f_{e_1} = \{\mathbf{f}\}$ From procedure BOR-CSET.
$S^t_{e_2} = \{(\mathbf{t},\mathbf{t},\mathbf{f}),(\mathbf{f},\mathbf{t},\mathbf{t})\}$ From procedure MI-CSET.
$S^f_{e_2} = \{(\mathbf{f},\mathbf{t},\mathbf{f}),(\mathbf{t},\mathbf{f},\mathbf{t})\}$ From procedure MI-CSET.

Now apply Step 4 to the constraints sets generated for subexpressions e_1 an e_2 and obtain the BOR constraint set for the entire expression E. This process is illustrated in Figure 4.14. The sets S^t_E, S^f_E, and S_E are listed below.

$$\begin{aligned}
S^t_E &= S^t_{e_1} \otimes S^t_{e_2} \\
&= \{(\mathbf{t},\mathbf{t},\mathbf{t},\mathbf{f}),(\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{t})\\
S^f_E &= (S^f_{e_1} \times \{t_2\}) \cup (\{t_1\} \times S^f_{e_2}) \\
&= \{(\mathbf{f},\mathbf{t},\mathbf{t},\mathbf{f}),(\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{f}),(\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{t})\} \\
S_E &= \{(\mathbf{t},\mathbf{t},\mathbf{t},\mathbf{f}),(\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{t}),(\mathbf{f},\mathbf{t},\mathbf{t},\mathbf{f}),(\mathbf{t},\mathbf{f},\mathbf{t},\mathbf{f}),(\mathbf{t},\mathbf{t},\mathbf{f},\mathbf{t})\}
\end{aligned}$$

Notice that constraint set S_E derived using the BOR-MI-CSET procedure contains only five constraints as compared to the six constraints in S_E derived using the MI-CSET procedure in Example 4.17. It is generally the case that the constraint sets derived using the BOR-MI-CSET procedure are smaller, and more effective in detecting faults, than those derived using the MI-CSET procedure. Exercises 4.23 and 4.24 should help you sharpen your understanding of the BOR-MI-CSET procedure and the effectiveness of the tests generated.

4.4.7 Cause-effect graphs and predicate testing

Cause-effect graphing, as described in Section 4.3 is a requirements modeling and test generation technique. Cause-effect relations are extracted from the requirements. Each "cause" portion of a cause-effect relation can

Figure 4.14 Constraint set for predicate $a(bc+\bar{b}d)$ derived using the BOR-MI-CSET procedure.

be expressed as a predicate. The "effect" portion is used to construct the oracle which decides if the effect does occur when the corresponding cause exists.

Empirical studies have revealed that the BOR-CSET procedure generates smaller test sets than the cause-effect graphing technique. However, note that the BOR method only considers predicates and not their effects.

To test if the condition representing the cause in a cause-effect graph has been implemented correctly, one could generate tests using either the decision table technique described in Section 4.3.3 or one of the four procedures described in this section.

Several studies have revealed that the BOR-CSET procedure generates significantly smaller tests sets than the CEGDT procedure described in Section 4.3.3. The fault detection effectiveness of the tests generated using the BOR-CSET procedure is only slightly less than that of tests generated using the CEGDT procedure.

The combination of cause-effect graphs and predicate testing procedures described in this section is considered useful for two reasons. First, the cause-effect graphs are a useful tool for modeling requirements. Second, once the graphical model has been built, any one of the four predicate-based test generation procedures introduced in this section can be applied for test generation. Exercise 4.25 is to help you understand how the combination of cause-effect graphs and predicate testing works.

4.4.8 Fault propagation

We now explain the notion of *fault propagation* that forms the basis of the four predicate test generation algorithms described in this section. Let p_r be a predicate, simple or compound, and p_c a component of p_r. Each node in the abstract syntax tree of a predicate is a component of that predicate. For example, as in Figure 4.13, predicate $p : (a+b) < c \wedge \neg p \vee r > s$ contains the following six components: $a+b<c$, $\wedge$, $\neg$, p, $\vee$, and $r>s$.

A test for an incorrect predicate p is considered error revealing if the evaluation of p against t leads to a value different from that obtained by evaluating the correct version of p against t.

Let p_f be a faulty version of p_r. We say that a fault in any component of p_f propagates to the root node of $AST(p_f)$, i.e. effects the outcome of p_f, if $p(t) \neq p_f(t)$ for some test case t. Test t is also known as an *error revealing* or *fault revealing* test for p_f.

In predicate testing we are interested in generating at least one test case t that ensures the propagation of the fault in p_f. The BOR, BRO, and BRE adequate test sets are very likely to guarantee certain kinds of faults, mentioned in earlier sections, propagate to the root node of the predicate under test. The next example illustrates the notion of fault propagation.

Example 4.19 Let $p : (a+b) < c \wedge \neg p \vee r > s$ and $p_f : (a+b) < c \vee \neg p \vee r > s$ be two predicates where p_f is a faulty version of p_r. p_f has a Boolean operator fault created by the incorrect use of the $\vee$

Figure 4.15 Abstract syntax trees in (a) and (b) show the failure of test t_1 to force the propagation of the $\wedge$ operator fault. (c) and (d) show how test t_4 ensures the propagation of the $\wedge$ operator fault.

operator. Table 4.3 lists four test cases that form a BOR-adequate test set.

Figure 4.15(a) and (b) show that test t_1 in Table 4.3 fails to cause the $\wedge$ operator fault to propagate to the root node of $AST(p_f)$ and thus we have $p(t_1) = p_f(t_1)$. However, as shown in Figure 4.15(c) and (d), test t_4 does cause the fault to propagate to the root node of $AST(p_f)$ and thus we have $p(t_4) \neq p_f(t_4)$.

Given a predicate p_r containing n AND/OR operators and Boolean variables, one can use a brute force method to generate a test set T with 2^n tests. For example, predicate $p : (a + b) < c \wedge \neg p \vee r > s$ contains two Boolean AND/OR operators and one Boolean variable; thus $n = 3$. A test set T generated in a brute force manner will contain a total of eight tests cases that satisfy the following set S of eight constraints.

BOR, BRO, and BRE adequate test sets are generally smaller than those derived by enumerating all combinations of the truth values of the components of a compound predicate.

$$S = \{(\mathbf{f}, \mathbf{f}, \mathbf{f}), (\mathbf{f}, \mathbf{f}, \mathbf{t}), (\mathbf{f}, \mathbf{t}, \mathbf{f}), \{(\mathbf{f}, \mathbf{t}, \mathbf{t}), (\mathbf{t}, \mathbf{f}, \mathbf{f}), (\mathbf{t}, \mathbf{f}, \mathbf{t}), (\mathbf{t}, \mathbf{t}, \mathbf{f}), (\mathbf{t}, \mathbf{t}, \mathbf{t})\}$$

These eight constraints ensure that each relational expression and Boolean variable in p_r will evaluate to `true` and `false` against at least one test case in T. Thus, T is obviously BOR, BRO, and BRE adequate. However, T is not minimal. Notice that while T contains eight test cases, a BRE adequate test set derived in Example 4.16 contains only five test cases.

A BOR, BRO, or BRE adequate test set contains 2*n+3 tests, where n is the number of AND or OR operators.

It can be shown that if a predicate p_r contains n AND/OR operators, then the maximum size of the BOR adequate test set is $n + 2$. The maximum size of a BRO or BRE adequate test set for p_r is $2 * n + 3$ (see Exercise 4.18).

As indicated above, the size of BOR, BRO, and BRE adequate tests grows linearly in the number of AND/OR and Boolean variables in a predicate. It is this nice property of the test generation procedures introduced in this section, in addition to fault detection effectiveness, that distinguishes them from the brute force method and the cause-effect graph-based test generation introduced earlier in Section 4.3.

The linearity property mentioned above is the result of (a) using the abstract syntax tree of a predicate to propagate the constraints from leaves to the root node and (b) using the **onto** ($\otimes$) product of two sets, instead of the set product ($\times$), while propagating the constraints up through the abstract syntax tree (see Exercise 4.19).

4.4.9 Predicate testing in practice

Predicate testing is used to derive test cases both from the requirements and from the application under test. Thus it can be used for generating *specification*-based and *program*-based tests.

Specification-based predicate test generation

An analysis of application requirements reveals conditions under which the application must behave in specific ways. Thus, an analysis of application requirements might yield the following list of n conditions and the associated tasks.

Condition	Task
C_1	$Task_1$
C_2	$Task_2$
$\vdots$	
C_n	$Task_n$

Such a list can be used to generate a test set T for the application under test. Note that each condition is represented by a predicate. The condition-task

pairs in the list above are assumed independent in the sense that each task depends exclusively on the condition associated with it. The independence assumption does not hold when, for example, a condition, say C_2, depends on C_1 in the sense that C_2 cannot be true if C_1 is true. For example, two such dependent conditions are: $C_1 : a < b$ and $C_2 : a > b$. Obviously, if $Task_1$ and $Task_2$ are associated exclusively with conditions C_1 and C_2, respectively, then only one of them can be performed for a given input.

As conditions can be compound and tasks can be conjoined, the independence requirement does not exclude the specification of two tasks to be performed when a condition is true, or that one task requires two conditions to hold. Two tasks could be conjoined to form one tasks corresponding to a condition. Similarly, two conditions could be combined to form a single condition associated with a single task.

For test generation, one first selects an appropriate test generation algorithm described in this section. Different algorithms could be used for different conditions. For example, if condition C_2 is represented by a non-singular predicate, then the BOR-MI-CSET algorithm will likely be preferred.

Once the test generation procedure is selected for a predicate, one generates a distinct test set. The union of the test sets so generated is a consolidated test set T for the application. It is certainly possible that two conditions yield the same test set.

Testing the application against test cases in T guarantees the detection of certain faults, indicated in Section 4.4.1, due to the incorrect coding of conditions from which the tests are derived. However, such a guarantee is valid only under certain conditions. The next example illustrates why a test set might fail.

Example 4.20 Suppose that we are to test system X. Requirements for X indicate that task $Task_1$ is to be performed when condition $C_1 : a < b \wedge c > d$ is true. Tests for C_1 are derived in Example 4.13 and are guaranteed to detect any Boolean operator fault in C_1.

Now consider an implementation of X whose structure is abstracted in Figure 4.16. Statement S_1 contains a fault in the coding of C_1. The condition coded is $C_f : a < b \vee c > d$. Let us assume that S_1 is unreachable for any test input.

Suppose that statement S_2 has been added by mistake, it should not be in the program. The condition C in S_2 is $C \wedge u$, where u is a Boolean variable input to X. Now suppose that u is set to true while testing X to check for the correct execution of $Task_1$.

Figure 4.16 An implementation to illustrate why a predicate test that is theoretically guaranteed to detect any Boolean operator fault, might fail to do so in practice.

A test set that theoretically is guaranteed to detect an error in a predicate might fail to do so in practice. This could happen, for example, due to an infeasible path.

The first three rows in the table below show that the observed behavior of X for each of the tests derived using the BOR-CSET procedure for condition C is identical to the expected behavior. However, when X is tested with $u = \texttt{false}$ and the BOR constraint $(\mathbf{t}, \mathbf{t})$, the error is revealed.

Test		Observed behavior	Expected behavior
u	$a < b \wedge c > d$		
`true`	$(\mathbf{t}, \mathbf{t})$	Execute $Task_1$	Execute $Task_1$.
`true`	$(\mathbf{t}, \mathbf{f})$	$Task_1$ not executed.	Do not execute $Task_1$.
`true`	$(\mathbf{f}, \mathbf{t})$	$Task_1$ not executed.	Do not execute $Task_1$.
`false`	$(\mathbf{t}, \mathbf{t})$	$Task_1$ not executed.	Execute $Task_1$.

Thus we have shown an instance where a test generated using the BOR-CSET procedure is unable to reveal the error. Certainly, this example is not demonstrating any weakness in the error detection effectiveness of the BOR-CSET procedure. Instead, it demonstrates that some errors in the program might mask an error in the implementation of a predicate.

Program-based predicate test generation

Tests can be derived for predicates obtained from the requirements or from the code.

Test sets can also be derived from predicates in the program. Test sets so derived are often referred to as *white box tests*. When executing such tests, one needs to ensure that the flow of control reaches the region in the program that implements the predicate being tested. As illustrated in the next example, this might cause some difficulty as several other conditions may need to be satisfied for the control to reach the desired region in the program.

Example 4.21 Consider the following program that uses two conditions to control the execution of three tasks named task1, task2, and task3. Using inputs r and e, the program determines values of parameters a, b, c, and d.

Program P4.1

```
1  begin
2   int a, b, c, d, r, e;
3    input (r, e);
4    getParam(r, e);
5   if(e < 0)
6   then
7     task1(a, b);
8   else
9      if(a < b&& c > d)
10   then
11    task2(c, d);
12  else
13   task3(c, d);
14 end
```

Suppose that the BOR-CSET procedure has been used to generate BOR-constraints from the predicate $a < b \&\& c > d$. Satisfying these constraints would be relatively easier if the parameters a, b, c, and d were program inputs. However, as is the case in our example program, these four parameters depend on the values of r and e. Thus, satisfying these constraints will require a determination of the relationship between r, e, and the four parameters.

Further, one must ensure that all tests of P4.1 that correspond to the constraints force the program to reach the statement containing the predicate, i.e. line 9. While this seems easy for our example program, it might not be so in most real-life programs.

4.5 Tests Using Basis Paths

Given a program P, one can find a set of basis paths that are linearly independent. All other paths in P can be derived as a linear combination of the basis paths. A path condition associated with a basis path serves as a source of tests.

Recall from Chapter 2 that a basis set is a set of basis paths derived from the CFG of a program. The basis paths are linearly independent complete paths. The following procedure can be used for generating tests for program P using basis paths.

1. Generate the CFG G from P.
2. Generate the basis set from G.
3. Do the following for each path in the basis set:
   (a) Find the path condition expressed in terms of the input variables.
   (b) Select values of the input variables that make the path condition true.
   (c) Determine the expected output for the input values selected above.

The above procedure will generate at least as many tests as there are basis paths. The next example illustrates the above process.

Example 4.22 Figure 4.17 shows the CFG of Program P3.1. The cyclomatic complexity of the CFG is $11 - 9 + 2 = 4$. The following four basis paths are derived from this CFG.

p_1 (`Start`, 1, 2, `End`)
p_2 (`Start`, 1, 3, 4, `End`)
p_3 (`Start`, 1, 3, 5, 6, `End`)
p_4 (`Start`, 1, 3, 5, 7, `End`)

The table below lists the path conditions, inputs that satisfy the conditions and expected response of the program. A "Don't care" entry indicates that the value could be arbitrary. File names "Exists", "DoesNotExist", and "Empty" indicate, respectively, a file that exists, does not exist, and is empty. For p_4 it is assumed that the given word appears four times in the input file.

Path	Word (w)	File (f)	Response
p_1	Don't care	DoesNotExist	Raise exception; return 0
p_2	""	Exists	Return 0
p_3	Don't care	Empty	Return 0
p_4	"Hello"	Exists	Return 4

Execution of the above tests requires the existence of two files, one that exists but is empty, and the other that exists and contains at

Figure 4.17 CFG for Program P4.1. Numbers against each node in the CFG correspond to the lines of code that belong to the corresponding basic block. (t) indicates the portion of the code executed when the condition at the indicated line is true. *t* and *f* label, respectively, the true and false branches.

least one occurrence of the given word. Processing of the exception raised by the first of the four tests above might lead to additional response items that are not shown above.

4.6 Scenarios and Tests

During system testing one often generates user-defined scenarios as tests. The assumption here is that a potential use of the system has an idea of how she wishes to use the system and hence can generate a few usage scenarios. Of course, a user in this case can also be the system tester.

A scenario describes a likely use of the systems features. It is a partial behavioral description of the system under test. Thus, one scenario is likely to test several features of a system. A scenario is generated based on the system requirements. It is assumed that the user of the tester generating scenarios knows what the requirements are.

A scenario may consist of events such as "ignition key is inserted," "ignition is turned ON," "gear is changed from neutral to first," and so on. In some cases a scenario might also consist of a sequence of activities where each activity is a sequence of events. Thus, a scenario becomes a kind of story line consisting of one or more events. The events could be natural,

A scenario is a story that describes the behavior of a system under certain constraints. User related scenarios derived from system requirements can be used to generate tests. Such kind of testing is useful when a complete system, or an application, is tested.

for example, earthquake, mechanical, for example, ignition turned ON, business, for example, order placed, or of any other kind.

An event-based scenario would be useful in testing a variety of devices, small or large such as a mobile phone or an automobile dashboard control software as well as business software, such as inventory management or finance management. A scenario is also useful in testing specific properties of a system. For example, a test scenario might be designed to check if the system can withstand a kind of attack such as denial of service or identity spoofing, or to test the performance of a system under certain conditions.

For a scenario to serve as a test case, each event in the scenario must be described explicitly. An event could be a user action or an input from another application in its environment.

For a scenario to be used as a test case, each event in the scenario must correspond to an observable response from the system under test. This leads to a scenario being a sequence of event-response pairs. Each event in the pair is input to the system manually or automatically. In the latter case some kind of a scenario simulator is needed that will create and apply the events, observe and analyze the system response to the event, and log it for future analysis.

Example 4.23 iDatify is a web-based tool for asking structured questions. Once created, a question can be made available to all users of the internet or to a specific community of people. Thus, two significant use-cases of iDatify are: question creation

Event/Activity	Response
Start iDatify	Login window visible.
Login	Home page for the user visible.
Select "Create community"	"New Community" form visible
Fill the form, select "Create"	Confirmation message displayed
Select "Ask a question"	"New Question" page visible
Add a question with response fields.	Create question visible
Select "Submit"	Question submitted confirmation message displayed.
Select "My communities"	Community just created is displayed.
Select "Ask a New Question"	"New Question "page visible
Add another question with response fields.	Create question visible
Select "Submit"	Question submitted confirmation message displayed.
Select "My communities"	Community just created is displayed.
Select the community just created	Community displayed. Both questions should be visible.

and community creation. Once a community has been created, questions can be created and associated with this community.

Below is a scenario to test the question and community creation features of iDatify. The scenario below is described in terms of user actions (events) and iDatify responses and may not be valid for any version of iDatify other than Version 0.23.

The underlined keywords above indicate an event or an activity. Notice that some activities, for example creating a question, are repeated. Repetition of events or activities within a scenario is intended to expose errors of initialization of variables.

Testing of complex system may require a large number of scenarios. Some of these scenarios ought to be aimed at testing the robustness of the system under test.

Testing of most complex systems will likely require hundreds or thousands of test scenarios. Manual execution of the test scenarios might be practically impossible as well as error prone. Thus, it is almost necessary to automate the process of scenario-based testing. For such systems, even the generation of scenarios is best done automatically from formal specifications such as finite state machines, UML diagrams, statecharts, and Z.

SUMMARY

In this chapter we have introduced another class of foundational techniques for generating tests. As in the previous chapter, the techniques in this chapter also partition the input domain, the focus is primarily the analysis of predicates to generate tests. A fault model for predicate testing is introduced. The BOR, BRO, and BRE techniques fall under the general category of syntax-based test generation. While these three techniques aim to detect the same types of errors as detected by the techniques mentioned above, the method used for test generation makes explicit use of the predicate syntax to generate tests and hence is quite different from the others.

Exercises

4.1 Consider the following variant of the program in Example 4.1.

```
String f(int x, int y){
  int q=0, z=0; String s=`` '';
  if(y<x+1) // Error 1: this should be y<x+2
   {s=s+``1'';  q=y*2;}
  if(q-x>0)
```

```
    {s=s+``2''; z=z+1;}
  if(z<=0) // Error 2:  this should be z<0
    s=s+``3'';
  s=s+``4'';
  return s;
}
```

Develop path condition for a path that goes through all three `if` statements. Generate tests as ON and OFF points for the path domain. Execute f against all tests and determine whether or not the errors are detected.

4.2 Consider the following incorrect function that contains a nonlinear path-condition.

```
int f(double x, double y){
  if(x*x<y) // Correct condition is  x*x<y*y
    return 1;
  else
    return 2;
}
```

Using the domain testing method develop test cases for the above function. Under what conditions will the generated tests detect the error?

4.3 An internet-enabled automatic Dog Food Dispenser, referred to as iDFD, consists of two separate dispensers: a food dispenser (FD) and a water dispenser (WD). Each dispenser is controlled by a separate timer which has a default setting of 8 hours and can also be set to a different period. Upon a timer interrupt the iDFD dispenses water or food depending on whether it is a water-timer or a food-timer interrupt. A fixed amount of food or water is dispensed.

The iDFD has two sets of three identical indicators, one for the food and the other for water. The indicators are labeled "Okay", "Low", and "Empty." After each dispensation, the iDFD checks the food and water levels and updates the indicators.

The iDFD can be connected to the internet via a secure connection. It allows an authorized user to set or reset the timers and also determine the water and food level in the respective containers. An email message is sent to the authorized dog sitter when food or water dispensation results in the corresponding indicators to show "Low" or "Empty."

The iDFD is controlled by an embedded microprocessor, and the iDFD control software, that is responsible for the correct operation of all the functions mentioned above. You are required to generate test inputs to test the control software in the following three steps:

(a) Identify all causes and effects.
(b) Relate the causes and effects through a cause-effect graph.
(c) Create a decision table from the cause-effect graph you have generated.
(d) Generate test inputs from the decision table.

While executing the steps above, you may need to resolve any ambiguities in the requirements. Also, consider separating the test inputs from water and food dispensers.

4.4 In Example 4.5 we ignored the effects of buyer selections on the GUI updates. For example, when a user selects CPU 3, the monitor window contents change from displaying all three monitors to displaying only monitor M 30. (a) Identify all such GUI related effects in the GUI-based computer purchase system in Example 4.5. (b) Draw a cause-effect graph that relates the causes given in the example to the effects you have identified. (c) Transform your cause-effect graph to a decision table. (d) Generate test data from the decision table.

4.5 Consider the four cause-effect graphs in Figure 4.18. (a) For each graph in the figure, generate combinations of input conditions (causes) that bring *Ef* to 1-state. Do not apply the heuristics given in Section 4.3.4. (b) Now reduce the combinations generated in (a) by applying the heuristics. How much reduction, measured in terms of the number of combinations discarded, do you achieve in each case? (c) Are the combinations generated in (b) unique in the sense that another set of combinations would also be satisfactory and as small in number?

Figure 4.18 Cause-effect graphs for Exercise 4.5.

4.6 Construct a sample implementation $\mathcal{I}$ to show that the application of heuristic H_2 from Table 4.1 will leave out at least one test that would have detected the error in $\mathcal{I}$.

4.7 Consider the following modified versions of heuristics H_1 and H_4 given in Table 4.1.

H_1: Enumerate any one combination of inputs to n_1 and n_2 such that $n_1 = n_2 = 0$.

H_4: Enumerate all combinations of inputs to n_1 and n_2 such that $n_1 = n_2 = 1$.

Discuss why the above modifications are undesirable (or desirable).

4.8 Complete Example 4.5 by (a) creating a decision table using procedure CEGDT; apply heuristics from Table 4.1 while generating the decision table. (b) Generate a set of tests from the decision table.

4.9 Complete Example 4.6 by reducing the combinations in the decision table using heuristics given in Table 4.1. Generate a set of tests given the following causes.

C_1: A user selects `Save As`.
C_2: A user selects `Open`.

C_3: A user types in an integer greater than 99 in the relevant window.
C_4: A user selects any one of the following values from a list: `High` or `Medium`.

4.10 Consider the following relational expression E_c: $2 * r < s + t$, where r, s, and t are floating point variables. Construct three faulty versions of E_c that contain, respectively, the off-by-ϵ, off-by-ϵ^*, and off-by-ϵ^+ faults. Select a suitable value for ϵ.

4.11 Consider the condition $C : (a + 1 > b) \wedge (c = d)$, where a, b, c, and d are type compatible variables. (a) Let $S_1 = \{(>,=),(>,>),(>,<),(=,=),(<,=)\}$ be a set of constraints on C. As in Example 4.11, construct a test set T_1 that satisfies S_1 and show that T is BRO-adequate. (b) Now consider the constraint set $S_2 = \{(+\epsilon,=),(=,-\epsilon),(-\epsilon,=),(+\epsilon,+\epsilon)\}$ on C. Construct a test set T_2 that satisfies S_2. Is T BRE adequate? Assume that $\epsilon = 1$.

4.12 Let $A = \{(<,=),(>,<)\}$ and $B = \{(\mathbf{t},=),(\mathbf{t},>),(\mathbf{f},<)\}$ be two constraint sets. Compute $A \times B$ and $A \otimes B$.

4.13 Consider the predicate $p_2 : (a+b) < c \wedge \neg p \vee (r > s)$ in Example 4.14. Confirm that any predicate obtained by inserting a Boolean operator fault in p_2 will evaluate to a value different from that of p_2 on at least one of the five test cases derived in Example 4.14.

4.14 Consider the predicate $p_r : (a+b) < c \wedge \neg p \vee (r > s)$ in Example 4.15. Confirm that any predicate obtained by inserting single or multiple Boolean operator and relational operator faults in p_r will evaluate to a value different from that of p_r on at least one of the four test cases derived in Example 4.15.

4.15 For predicate p_r, the BRO adequate test set T_{BRO} in Table 4.4 contains one more test case than the BOR adequate test set T_{BRO} in Table 4.3. (a) Is there a faulty predicate obtained from p_r that will be distinguished from p_r by T_{BRO} and not by T_{BOR}? (b) Is there a faulty predicate obtained from p_r that will be distinguished from p_r by T_{BOR} and not by T_{BRO}? Constrain your answer to the fault model in Section 4.4.1.

4.16 Consider a predicate p_r and test sets T_{BRO} and T_{BRE} that are, respectively, BRO and BRE-adequate for p_r. For what value of ϵ are T_{BRO} and T_{BRE} equivalent in their fault detection abilities?

4.17 (a) Show that a BRE constraint set can derived from a BRO constraint set simply by replacing constraint $(>)$ by $(+\epsilon)$, $(<)$ by $(-\epsilon)$ and retaining $(=)$. (b) Explain why a BRO-adequate test set might not be BRE-adequate. (c) Is a BRE-adequate test set always BRO-adequate? BOR-adequate? (d) Is a BRO-adequate test set always BOR-adequate?

4.18 Let p_r be a predicate that contains at most n AND/OR operators. Show that $|T_{BOR}| \leq n + 2$, $|T_{BRO}| \leq 2 * n + 3$, and $|T_{BRE}| \leq 2 * n + 3$.

4.19 Procedures BOR-CSET, BRO-CSET, and BRE-CSET use the $\otimes$ operator while propagating the constraints up through the abstract syntax tree. Show that larger test sets would be generated if instead the $\times$ operator is used.

4.20 What conflict(s) might arise when applying any of procedures BOR-CSET, BRO-CSET, and BRE-CSET to a predicate containing one or more non-singular expressions? Answer this question using predicate $p_r : (a+b)(bc)$ and applying any of the three procedures.

4.21 Use the BOR-CSET procedure to generate tests from the predicate $p_r : a+b$. (a) Do the tests generated guarantee the detection of the missing Boolean variable fault that causes the implementation of p_r to be a? (b) Do the tests generated guarantee the detection of the extra Boolean variable fault in coding p_r that causes the implementation to be $a+b+c$, where c is the extra Boolean variable ?

4.22 In procedure MI-CSET in Section 4.4.6, show that (a) $S^t_{e_i} \neq \emptyset$ for $1 \leq i \leq n$ and (b) $FS_{e^j_i}$ for different terms in p_r may not be disjoint and may be empty.

4.23 Use the constraint set S_E derived in Example 4.18 to show that the following faulty versions of the predicate $E = a(bc+\overline{b}d)$ evaluate differently than E on at least one test case derived from S_E.

(i)	$a(bc+bd)$	Missing NOT operator.
(ii)	$a(\overline{b}c+\overline{b}d)$	Incorrect NOT operator.
(iii)	$a+(bc+\overline{b}d)$	Incorrect OR operator.
(iv)	$a(bc\overline{b}d)$	Incorrect AND operator.
(iii)	$a+(\overline{b}c+\overline{b}d)$	Incorrect OR and NOT operators.

4.24 (a) Use the BOR-MI-CSET procedure to derive the constraint set for the following predicate p_r.

$$(a<b) \wedge (((r>s) \wedge u) \vee (a \geq b) \vee ((c<d) \vee (f=g)) \wedge (v \wedge w)),$$

where a, b, c, d, f, g, r, and s are integer variables and u, v, and w are Boolean variables.
(b) Using the constraint set derived in (a), construct a test set T for p_r. (c) Verify that T is able to distinguish the following faulty versions from p_r.

(*i*) $(a<b) \wedge ((\neg(r>s) \wedge u) \vee (a \geq b) \vee ((c<d) \vee (f=g)) \wedge (v \wedge w))$; Incorrect NOT operator.
(*ii*) $(a<b) \wedge (((r>s) \wedge u) \vee (a \geq b) \vee ((c<d) \vee (f=g)) \wedge (\overline{v} \wedge w)$; Incorrect NOT operator.
(*iii*) $(a<b) \vee (((r>s) \vee u) \vee (a \geq b) \vee ((c<d) \vee (f=g)) \wedge (v \wedge w))$; Two incorrect OR operators.
(*iv*) $(a<b) \vee (((r>s) \wedge \overline{u}) \vee (a \geq b) \vee ((c<d) \vee (f=g)) \wedge (v \wedge w))$; Incorrect OR and NOT operators.

4.25 Consider the cause-effect graph shown in Figure 4.19 that exhibits the relationship between several causes and effect E.

(a) Derive a test set T_{CEG} to test the implementation of E using the test generation procedure described in Section 4.3.3.

Figure 4.19 A cause-effect graph for Exercise 4.25. Variables a, b, c and d are integers, $u : a < b$, $v : c > d$, and w is a Boolean variable.

(b) Construct a Boolean expression p_r from the Figure 4.19 that represents the condition for the effect to occur. Select one of the four procedures BOR-CSET, BRO-CSET, BRE-CSET, and BRO-MI-CSET, and apply it to construct a test set from p_r. Assume that there might be errors in the relational expressions that represent various causes effecting E. Construct a test set T_B from the constraints you derive.

(c) Suppose that u is incorrectly implemented as $a < b + 1$. Does T_{CEG} contains an error-revealing test case for this error? How about T_B?

4.26 Construct at least two different scenarios that capture the behavior of a user of an Automated Teller Machine (ATM).

4.27 Construct a program where testing a feature twice, without restarting the program, will lead to failure.

4.28 Give one example of a situation where a new scenario can be generated by combining two existing scenarios. Suggest a few different ways of combining scenarios.

5

Test Generation from Finite State Models

CONTENTS

5.1 Software design and testing
5.2 Finite state machines
5.3 Conformance testing
5.4 A fault model
5.5 Characterization set
5.6 The W-method
5.7 The partial W-method
5.8 The UIO-sequence method
5.9 Automata theoretic versus control-flow based techniques
5.10 Tools

The purpose of this chapter is to introduce techniques for the generation of test data from finite state models of software designs. A fault model and three test generation techniques are presented. The test generation techniques presented are the W-method, the Unique Input/Output method, and the Wp-method.

5.1 Software Design and Testing

Development of most software systems includes a design phase. In this phase the requirements serve as the basis for a design of the application to be developed. The design itself can be represented using one or more notations. Finite state machines (FSM), statecharts, and Petri nets are three design notations used in software design.

A simplified software development process is shown in Figure 5.1. Requirements analysis and design is a step that ideally follows requirements gathering. The end result of the design step is a *design* artifact that expresses a high-level application architecture and interactions amongst low-level application components. The design is usually expressed using a variety of notations such as those embodied in the UML design language. For example, UML statecharts are used to represent the design of the real-time portion of the application, and UML sequence diagrams are used to show the interactions between various application components.

Tests can be generated directly from formal expressions of software designs. Such tests can be used to test an implementation against its design.

The design is often subjected to test prior to its input to the coding step. Simulation tools are used to check if the state transitions depicted in the statecharts do conform to the application requirements. Once the

Figure 5.1 Design and test generation in a software development process. A *design* is an artifact generated in the Analysis and Design phase. This artifact becomes an input to the Test Generator algorithm that generates tests for input to the code during testing.

design has passed the test, it serves as an input to the coding step. Once the individual modules of the application have been coded, successfully tested, and debugged, they are integrated into an application and another test step is executed. This test step is known by various names such as *system test* and *design verification test*. In any case, the test cases against which the application is run can be derived from a variety of sources, design being one.

In this chapter, we show how a design can serve as source of tests that are used to test the application itself. As shown in Figure 5.1, the design generated at the end of the analysis and design phase serves as an input to a *test generation* procedure. This test generation procedure generates a number of tests that serve as inputs to the code in the test phase. Note that though Figure 5.1 seems to imply that the test generation procedure is applied to the entire design, this is not necessarily true; tests can be generated using portions of the design.

Finite State Machines (FSMs) and Statecharts are two well-known formalisms for expression software designs. Both these formalisms are part of UML.

We introduce several test generation procedures to derive tests from FSMs. An FSM offers a simple way to model state-based behavior of applications. The statechart is a rich extension of the FSM and needs to be handled differently when used as an input to a test generation algorithm. The Petri net is a useful formalism to express concurrency and timing and leads to yet another set of algorithms for generating tests. All test generation methods described in this chapter can be automated, though only some have been integrated into commercial test tools.

There exist several algorithms that take an FSM and some attributes of its implementation as inputs to generate tests. Note that the FSM serves as a source for test generation and is not the item under test. It is the implementation of the FSM that is under test. Such an implementation is also known as *Implementation Under Test* and abbreviated as IUT. For example, an FSM may represent a model of a communication protocol while an IUT is its implementation. The tests generated by the algorithms introduced in this chapter are inputs to the IUT to determine if the IUT behavior conforms to the requirements.

The following methods are introduced in this chapter: the W-method, the transition tour method, the distinguishing sequence method, the unique input/output method, and the partial W-method. In Section 5.9, we compare the various methods introduced. Before proceeding to describe the test generation procedures, we introduce a fault model for FSMs, the characterization set of an FSM, and a procedure to generate this set. The characterization set is useful in understanding and implementing the test generation methods introduced subsequently.

5.2 Finite State Machines

Many devices used in daily life contain embedded computers. For example, an automobile has several embedded computers to perform various tasks, engine control being one example. Another example is a computer inside a toy for processing inputs and generating audible and visual responses. Such devices are also known as *embedded systems*. An embedded system can be as simple as a child's musical keyboard or as complex as the flight controller in an aircraft. In any case, an embedded system contains one or more computers for processing inputs.

An FSM captures the behavior of a software or a software-hardware system as a finite sequence of states and transitions among them. The behavior of the modeled system for a given sequence of inputs can be obtained by starting the machine in its initial state and applying the sequence. This activity is also known as simulating the FSM.

An embedded computer often receives inputs from its environment and responds with appropriate actions. While doing so, it moves from one state to another. The response of an embedded system to its inputs depends on its current state. It is this behavior of an embedded system in response to inputs that is often modeled by an FSM. Relatively simpler systems, such as protocols, are modeled using FSMs. More complex systems, such as the engine controller of an aircraft, are modeled using *statecharts* which can be considered as a generalization of FSMs. In this section, we focus on FSMs. The next example illustrates a simple model using an FSM.

Example 5.1 Consider a traditional table lamp that has a three-way rotary switch. When the switch is turned to one of its positions, the lamp is in the OFF state. Moving the switch clockwise to the next position moves the lamp to an ON-DIM state. Moving the switch further clockwise moves the lamp to the ON-BRIGHT state. Finally, moving the switch clockwise one more time brings the lamp back to the OFF state. The switch serves as the input device that controls the state of the lamp. The change of lamp state in response to the switch position is often illustrated using a *state* diagram as in Figure 5.2(a).

Our lamp has three states OFF, ON_DIM, and ON_BRIGHT, and one input. Note that the lamp switch has three distinct positions, though from a lamp user's perspective the switch can only be turned clockwise to its next position. Thus, "turning the switch one notch clockwise" is the only input. Suppose that the switch also can be moved counterclockwise. In this latter case, the lamp has two inputs one corresponding to the clockwise motion and the other to the counterclockwise motion. The number of distinct states of the lamp remains three but the state diagram is now as in Figure 5.2(b).

Figure 5.2 Change of lamp state in response to the movement of a switch. (a) Switch can only move one notch clockwise (CW). (b) Switch can move one notch clockwise and counterclockwise (CCW).

In the above example, we have modeled the table lamp in terms of its states, inputs, and transitions. In most practical embedded systems, the application of an input might cause the system to perform an *action* in addition to performing a state transition. The action might be as trivial as "do nothing" and simply move to another state, or a complex function might be executed. The next example illustrates one such system.

We assume that upon the application of an input, an FSM moves from its current state to its next state and generates some observable output. The output could be, for example, change of screen, change of file contents, a message sent on the network, etc.

Example 5.2 Consider a machine that takes a sequence of one or more unsigned decimal digits as input and converts the sequence to an equivalent integer. For example, if the sequence of digits input is 3, 2, and 1, then the output of the machine is 321. We assume that the end of the input digit sequence is indicated by the asterisk (*) character. We shall refer to this machine as the DIGDEC machine. The state diagram of DIGDEC is shown in Figure 5.3.

The DIGDEC machine can be in any one of three states. It begins its operation in state q_0. Upon receiving the first digit, denoted by `d` in the figure, it invokes the function INIT to initialize variable `num` to `d`. In addition, the machine moves to state q_1 after performing

Figure 5.3 State diagram of the DIGDEC machine that converts sequence of decimal digits to an equivalent decimal number.

the INIT operation. In q_1 DIGDEC can receive a digit or an end-of-input character which in our case is the asterisk. If it receives a digit, it updates `num` to $10 * \texttt{num} + d$ and remains in state q_1. Upon receiving an asterisk the machine executes the OUT operation to output the current value of `num` and moves to state q_2. Notice the double circle around state q_2. Often, such a double circle is used to indicate a *final* state of an FSM.

FSMs are found in two types: Moore machines and Mealy machines. Moore machines do not associate actions with transitions while Mealy machines do.

Historically, FSMs that do not associate any action with a transition are known as *Moore* machines. In Moore machines, actions depend on the current state. In contrast, FSMs that do associate actions with each state transition are known as *Mealy* machines. In this book, we are concerned with Mealy machines. In either case, an FSM consists of a finite set of states, a set of inputs, a start state, and a transition function which can be defined using the state diagram. In addition, a Mealy machine has a finite set of outputs. A formal definition of a Mealy machine follows.

Finite State Machine:

A finite state machine is a six tuple $(X, Y, Q, q_0, \delta, O)$ *where*

- X is a finite set of input symbols also known as the *input alphabet*,
- Y is a finite set of output symbols also known as the *output alphabet*,
- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- $\delta : Q \times X \rightarrow Q$ is a next-state or state transition function, and
- $O : Q \times X \rightarrow Y$ is an output function.

In some variants of FSM more than one state could be specified as an initial state. Also, sometimes, it is convenient to add $F \subseteq Q$ as a set of *final* or *accepting* states while specifying an FSM. The notion of accepting states is useful when an FSM is used as an automaton to recognize a language. Also note that the definition of the transition function δ implies that for any state q_i in Q, there is at most one next state. Such FSMs are also known as *deterministic* FSMs. In this book, we are concerned only with deterministic FSMs. The state transition function for non-deterministic FSMs is defined as

$$\delta : Q \times X \rightarrow 2^Q$$

which implies that such an FSM can have more than one possible transition out of a state on the same symbol. Non-deterministic FSMs are

Figure 5.4 Multiple labels for an edge connecting two states in an FSM.

usually abbreviated as NFSM or NDFSM or simply as NFA for Non-deterministic Finite Automata.

The state transition and the output functions are extended to strings as follows. Let q_i and q_j be two, possibly same, states in Q and $s = a_1 a_2 \ldots a_n$ a string over the input alphabet X of length $n \geq 0$. $\delta(q_i, s) = q_j$ if $\delta(q_i, a_1) = q_k$ and $\delta(q_k, a_2 a_3 \ldots a_n) = q_j$. The output function is extended similarly. Thus, $O(q_i, s) = O(q_i, a_1) \cdot O(\delta(q_i, a_1), a_2, a_3, \ldots, a_n)$. Also, $\delta(q_i, \epsilon) = q_i$ and $O(q_i, \epsilon) = \epsilon$.

A non-deterministic FSM (NDFSM) is one in which there are multiple outgoing transitions from state on the same input. Every NDFSM can be converted into an equivalent deterministic FSM.

Table 5.1 contains a formal description of the state and output functions of FSMs given in Figures 5.2 and 5.3. There is no output function for the table lamp. Note that the state transition function δ and the output function O can be defined by a *state diagram* as in Figures 5.2 and 5.3. Thus, for example, from Figure 5.2(a), we can derive δ(OFF, Turn_CW) = ON_DIM. As an example of the output function, we get O = INIT(num, 0) from Figure 5.3.

A state diagram is a directed graph that contains nodes representing states and edges representing state transitions and output functions. Each node is labeled with the state it represents. Each directed edge in a state diagram connects two states.

Each edge is labeled i/o where i denotes an input symbol that belongs to the input alphabet X and o denotes an output symbol that belongs to the

The transition function of an FSM can be described using a state diagram or a table.

Table 5.1 Formal description of three FSMs from Figures 5.2 and 5.3.

	Figure 5.2 (a)	**Figure 5.2 (b)**	**Figure 5.3**
X	{Turn-CW}	{Turn-CW, Turn-CCW}	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, *}
Y	None	None	{INIT, ADD, OUT}
Q	{OFF, ON-DIM, ON-BRIGHT}	{OFF, ON-DIM, ON-BRIGHT}	$\{q_0, q_1, q_2\}$
q_0	{OFF}	{OFF}	q_0
F	None	None	q_2
δ	See Figure 5.2(a)	See Figure 5.2(b)	See Figure 5.3
O	Not applicable	Not applicable	See Figure 5.3

output alphabet Y. i is also known as the *input portion* of the edge and o its *output portion*. Both i and o might be abbreviations. As an example, the edge connecting states q_0 and q_1 in Figure 5.3 is labeled $d/INIT(num, d)$ where d is an abbreviation for any digit from 0 to 9 and INIT(num,d) is an action.

Multiple labels can also be assigned to an edge. For example, the label i_1/o_1 and i_2/o_2 associated with an edge that connects two states q_i and q_j implies that the FSM can move from q_i to q_j upon receiving either i_1 or i_2. Further, the FSM outputs o_1 if it receives input i_1 and outputs o_2 if it receives o_2. The transition and the output functions can also be defined in tabular form as explained in Section 5.2.2.

5.2.1 Excitation using an input sequence

In most practical situations, an implementation of an FSM is excited using a sequence of input symbols drawn from its input alphabet. For example, suppose that the table lamp whose state diagram is in Figure 5.2(b) is in state ON-BRIGHT and receives the input sequence r where

$r =$ Turn_CCW Turn_CCW Turn_CW

Using the transition function in Figure 5.2(b), it is easy to verify that the sequence s will move the lamp from state ON_BRIGHT to state ON_DIM. This state transition can also be expressed as a sequence of transitions given below.

δ(ON_BRIGHT, Turn_CCW)= ON_DIM
δ(ON_DIM, Turn_CCW)= OFF
δ(OFF, Turn_CW)= ON_DIM

For brevity, we will use the notation $\delta(q_k, z) = q_j$ to indicate that an input sequence z of length one or more moves an FSM from state q_k to state q_j. Using this notation, we can write δ(ON_BRIGHT, r) = ON_DIM for the state diagram in Figure 5.2(b).

We will use a similar abbreviation for the output function O. For example, when excited by the input sequence 1001*, the DIGDEC machine ends up in state q_2 after executing the following sequence of actions and state transitions:

$O(q_0, 1) =$ INIT(num,1), $\delta(q_0, 1) = q_1$
$O(q_1, 0) =$ ADD(num,0), $\delta(q_1, 0) = q_1$
$O(q_1, 0) =$ ADD(num,0), $\delta(q_1, 0) = q_1$

$O(q_1, 1) = \text{ADD(num,1)}, \delta(q_1, 1) = q_1$
$O(q_1, *) = \text{OUT(num)}, \delta(q_1, *) = q_2$

Once again, for brevity, we will use the notation $O(q, r)$ to indicate the action sequence executed by the FSM on input r when starting in state q. We also assume that a machine remains in its current state when excited with an empty sequence. Thus, $\delta(q, \epsilon) = q$. Using the abbreviation for state transitions, we can express the above action and transition sequence as follows

$$O(q_0, 1001*) = \text{INIT(num,1) ADD(num,0) ADD(num,0) ADD(num,1) OUT(num)}$$
$$\delta(q_0, 1001*) = q_2$$

5.2.2 Tabular representation

A table is often used as an alternative to the state diagram to represent the state transition function δ and the output function O. The table consists of two sub-tables that consist of one or more columns each. The leftmost sub-table is the *output* or the *action* sub-table. The rows are labeled by the states of the FSM. The rightmost sub-table is the *next state* sub-table. The output sub-table, that corresponds to the output function O, is labeled by the elements of the input alphabet X. The next state sub-table, that corresponds to the transition function δ, is also labeled by elements of the input alphabet. The entries in the output sub-table indicate the actions taken by the FSM for a single input received in a given state. The entries in the next state sub-table indicate the state to which the FSM moves for a given input and in a given current state. The next example illustrates how the output and transition functions can be represented in a table.

The tabular representation of the transition function of an FSM is convenient for a computer implementation.

Example 5.3 The table given below shows how to represent functions δ and O for the DIGDEC machine. In this table, the column labeled "Action" defines the output function by listing the actions taken by the DIGDEC machine for various inputs. Sometimes, this column is also labeled as "Output" to indicate that the FSM generates an output symbol when it takes a transition. The column labeled "Next State" defines the transition function by listing the next state of the DIGDEC machine in response to an input. Empty entries in the table should be considered as undefined. However, in certain practical situations, the empty entries correspond to an "error" action. Note that in the table below we have used the abbreviation d for the ten digits, 0 through 9.

Current state	Action		Next state	
	d	*	d	*
q_0	INIT (num, d)		q_1	
q_1	ADD (num, d)	OUT (num)	q_1	q_2
q_2				

An FSM is complete, or completely specified, if there is exactly one transition in each state for every element in the input alphabet.

5.2.3 Properties of FSM

FSMs can be characterized by one or more of several properties. Some of the properties found useful in test generation are given below. We shall use these properties later while explaining algorithms for test generation.

Completely specified FSM: An FSM M is said to be *completely specified* if from each state in M there exists a transition for each input symbol. The machine described in Example 5.1 with state diagram in Figure 5.2(a) is completely specified as there is only one input symbol and each state has a transition corresponding to this input symbol. Similarly, the machine with state diagram shown in Figure 5.2(b) is also completely specified as each state has transitions corresponding to each of the two input symbols. The DIGDEC machine in Example 5.2 is not completely specified as state q_0 does not have any transition corresponding to an asterisk and state q_2 has no outgoing transitions.

An FSM is strongly connected if for every pair of states, say (q_1, q_2), there exists an input sequence that takes it from q_1 to q_2.

Strongly connected: An FSM M is considered *strongly connected* if for each pair of states (q_i, q_j), there exists an input sequence that takes M from state q_i to q_j. It is easy to verify that the machine in Example 5.1 is strongly connected. The DIGIDEC machine in Example 5.2 is not strongly connected as it is impossible to move from state q_2 to q_0, from q_2 to q_1, and from q_1 to q_0. In a strongly connected FSM, given some state $q_i \neq q_0$, one can find an input sequence $s \in X^*$ that takes the machine from its initial state to q_i. We therefore say that in a strongly connected FSM, every state is *reachable* from the initial state.

Two states in an FSM, say q_1 and q_2, are considered distinguishable if there exists at least one input sequence, say s, such that its application to the FSM in q_1 and in q_2 leads to different outputs. If no such s exists then q_1 and q_2 are considered equivalent.

V-equivalence: Let $M_1 = (X, Y, Q_1, m_0^1, T_1, O_1)$ and $M_2 = (X, Y, Q_2, m_0^2, T_2, O_2)$ be two FSMs. Let V denote a set of non-empty strings over the input alphabet X, i.e. $V \subseteq X^+$. Let q_i and q_j, $i \neq j$, be the states of machines M_1 and M_2, respectively. States q_i and q_j are considered V-equivalent if $O_1(q_i, s) = O_2(q_j, s)$ for all $s \in V$. Stated differently, states q_i and q_j are considered *V-equivalent* if M_1 and M_2, when excited in states q_i and q_j, respectively, yield identical output sequences. States q_i and q_j are said to be

equivalent if $O_1(q_i, s) = O_2(q_j, s)$ for any set V. If q_i and q_j are not equivalent, then they are said to be *distinguishable*. This definition of equivalence also applies to states within a machine. Thus, machines M_1 and M_2 could be the same machine.

Machine equivalence: Machines M_1 and M_2 are said to be equivalent if (a) for each state σ in M_1, there exists a state σ' in M_2 such that σ and σ' are equivalent and (b) for each state σ in M_2, there exists a state σ' in M_1 such that σ and σ' are equivalent. Machines that are not equivalent are considered distinguishable. If M_1 and M_2 are strongly connected, then they are equivalent if their respective initial states, m_0^1 and m_0^2, are equivalent. We write $M_1 = M_2$ if machines M_1 and M_2 are equivalent and $M_1 \neq M_2$ when they are distinguishable. Similarly, we write $q_i = q_j$ when states q_i and q_j are equivalent and $q_i \neq q_j$ if they are distinguishable.

Two FSMs are considered equivalent if they are indistinguishable. Such machines will generate identical outputs for each input sequence.

k-equivalence: Let $M_1 = (X, Y, Q_1, m_0^1, T_1, O_1)$ and $M_2 = (X, Y, Q_2, m_0^2, T_2, O_2)$ be two FSMs. States $q_i \in Q_1$ and $q_j \in Q_2$ are considered *k-equivalent* if when excited by any input of length k, yield identical output sequences. States that are not *k-equivalent* are considered *k-distinguishable*. Once again, M_1 and M_2 may be the same machines implying that k-distinguishability applies to any pair of states of an FSM. It is also easy to see that if two states are k-distinguishable for any $k > 0$, then they are also distinguishable for any $n > k$. If M_1 and M_2 are not *k-distinguishable*, then they are said to be *k-equivalent*.

Two FSMs are k-equivalent if no string of length k or less over its input alphabet can distinguish them. FSMs that can be distinguished by strings of length greater than k are considered k-distinguishable.

Minimal machine: An FSM M is considered *minimal* if the number of states in M is less than or equal to any other FSM equivalent to M.

An FSM M is considered minimal if there exists no other FSM with fewer states and defines the same output function as M.

Example 5.4 Consider the DIGDEC machine in Figure 5.3. This machine is not completely specified. However, it is often the case

Figure 5.5 State diagram of a completely specified machine for converting a sequence of one or more decimal digits to their decimal number equivalent. d|* refers to digit or any other character.

that certain erroneous transitions are not indicated in the state diagram. A modified version of the DIGDEC machine appears in Figure 5.5. Here, we have labeled explicitly the error transitions by the output function ERROR( ).

5.3 Conformance Testing

Testing an implementation against its design is also known as conformance testing.

The term *conformance testing* is used during the testing of communication protocols. An implementation of a communication protocol is said to conform to its specification if the implementation passes a collection of tests derived from the specification. In this chapter, we introduce techniques for generating tests to test an implementation for conformance testing of a protocol modeled as an FSM. Note that the term *conformance testing* applies equally well to the testing of any implementation that corresponds to its specification, regardless of whether or not the implementation is that of a communication protocol.

Communication protocols are used in a variety of situations. For example, a common use of protocols is in public data networks. These networks use access protocols such as the X.25 which is a protocol standard for wide area network communications. The Alternating Bit Protocol (ABP) is another example of a protocol used in the connection-less transmission of messages from a transmitter to a receiver. Musical instruments such as synthesizers and electronic pianos use the MIDI (Musical Instrument Digital Interface) protocol to communicate amongst themselves and a computer. These are just a few examples of a large variety of communication protocols that exist and new ones are often under construction.

FSM are used to specify a variety of systems- both physical and logical. Communication protocols can be specified using FSMs and can be the behavior of a robot.

A protocol can be specified in a variety of ways. For example, it could be specified informally in textual form. It could also be specified more formally using an FSM, a visual notation such as a statechart, and using languages such as LOTOS (Language of Temporal Ordering Specification), Estelle, and SDL. In any case, the specification serves as a source for implementers and also for automatic code generators. As shown in Figure 5.6, an implementation of a protocol consists of a control portion and a data portion. The control portion captures the transitions in the protocol from one state to another. It is often modeled as an FSM. The data portion maintains information needed for the correct behavior. This includes counters and other variables that hold data. The data portion is modeled as a collection of program modules or segments.

Testing an implementation of a protocol involves testing both the control and data portions. The implementation under test is often referred to as IUT. In this chapter, we are concerned with testing the control portion of an IUT. Techniques for testing the data portion of the IUT are described

Figure 5.6 Relationship amongst protocol specification, implementation, and models.

in other chapters. Testing the control portion of an IUT requires the generation of test cases. As shown in Figure 5.7, the IUT is tested against the generated tests. Each test is a sequence of symbols that are input to the IUT. If the IUT behaves in accordance with the specification, as determined by an oracle, then its control portion is considered equivalent to the specification. Non-conformance usually implies that the IUT has an error that needs to be fixed. Such tests of an IUT are generally effective in revealing certain types of implementation errors discussed in Section 5.4.

A significant portion of this chapter is devoted to describing techniques for generating tests to test the control portion of an IUT corresponding

A typical computer program contains both control and data. An FSM captures only the control portion of the program (or its design). Thus, tests generated using an FSM test for the correctness of the control portion of the implementation.

Figure 5.7 A simplified procedure for testing a protocol implementation against an FSM model. Italicized items represent data input to some procedure. Note that the model itself serves as data input to the test generator and as a procedure for deriving the expected output.

to a formally specified design. The techniques described for testing IUTs modeled as FSMs are applicable to protocols and other requirements that can be modeled as FSMs. Most complex software systems are generally modeled using statecharts, and not FSMs. However, some of the test generation techniques described in this chapter are useful in generating tests from statecharts.

5.3.1 Reset inputs

A reset input is needed to bring an implementation to its starting state. Such an input is useful in automated testing. A simple way to reset an implementation is to restart it. However, automated testing might be faster if instead of restarting an application, it can be brought to its initial state while the program resides in memory.

The test methods described in this chapter often rely on the ability of a tester to reset the IUT so that it returns to its start state. Thus, given a set of test cases $T = \{t_1, t_2, \ldots, t_n\}$, a test proceeds as follows:

1. Bring the IUT to its start state. Repeat the following steps for each test in T.
2. Select the next test case from T and apply it to the IUT. Observe the behavior of the IUT and compare it against the expected behavior as shown in Figure 5.7. The IUT is said to have failed if it generates an output different from the expected output.
3. Bring the IUT back to its start state by applying the reset input and repeat the above step but with the next test input.

It is usually assumed that the application of the reset input generates a `null` output. Thus, for the purpose of testing an IUT against its control specification FSM $M = (X, Y, Q, q_1, \delta, O)$, the input and output alphabets are augmented as follows:

$$X = X \bigcup \{Re\}$$
$$Y = Y \bigcup \{\texttt{null}\}$$

where Re denotes the reset input and `null` the corresponding output.

Side effects might make it difficult to bring an IUT to its start state. Such side effects include change in the environment such as a database or the state of any connected hardware on which the behavior of the IUT depends.

While testing a software implementation against its design, a reset input might require executing the implementation from the beginning, i.e. restarting the implementation, manually or automatically via, for example, a script. However, in situations where the implementation causes side effects, such as writing to a file or sending a message over a network, bringing the implementation to its start state might be a non-trivial task if its behavior depends on the state of the environment. While testing continuously running programs, such as network servers, the application of a reset input does imply bringing the server to its initial state but not necessarily shutting it down and restarting.

Figure 5.8 Transitions corresponding to reset (*Re*) inputs.

The transitions corresponding to the reset input are generally not shown in the state diagram. In a sense, these are hidden transitions that are allowed during the execution of the implementation. When necessary, these transitions could be shown explicitly as in Figure 5.8.

Example 5.5 Consider the test of an application that resides inside a microwave oven. Suppose that one test, say t_1, corresponds to testing the "set clock time" functions. Another test, say t_2, corresponds to the heating ability of the oven under "low power" setting. Finally, the third test t_3 tests the ability of the oven under "high power" setting. In its start state, the oven has electrical power applied and is ready to receive commands from its keypad. We assume that the clock time setting is not a component of the start state. This implies that the current time on the clock, e.g. 1:15 pm or 2:09 am, does not have any effect on the state of the oven.

It seems obvious that the three tests mentioned above can be executed in any sequence. Further, once a test is completed and the oven application, and the oven, does not fail, the oven will be in its start state and hence no explicit reset input is required prior to executing the next test.

However, the scenario mentioned above might change after the application of, for example, t_2, if there is an error in the application under test or in the hardware that receives control commands from the application under test. For example, the application of t_2 is expected to start the oven's heating process. However, due to some hardware or software error, this might not happen and the oven application might enter a loop waiting to receive a "task completed" confirmation signal from the hardware. In this case, tests t_3 or t_1 cannot be applied unless the oven control software and hardware are reset to their start state.

It is due to situations such as the one described in the previous example that require the availability of a reset input to bring the IUT under test to its start state. Bringing the IUT to its start state gets it ready to receive the next test input.

5.3.2 The testing problem

There are several ways to express the design of a system or a subsystem. FSM, statecharts, and Petri Nets are some of the formalisms to express various aspects of a design. Protocols, for example, are often expressed as FSMs. Software designs are expressed using statecharts. Petri nets are often used to express aspects of designs that relate to concurrency and timing. The design is used as a source for the generation of tests that are used subsequently to test the IUT for conformance to the design.

Given an FSM and an IUT, the testing problem is to determine if the IUT is equivalent to the FSM. While in general this problem is undecidable, assumptions on the nature of faults in the IUT and the behavior of the IUT itself make the problem tractable.

Let M be a formal representation of a design. As above, this could be an FSM, a statechart, or a Petri net. Other formalisms are possible too. Let *R* denote the requirements that led to M. *R* may be for a communication protocol, an embedded system such as the automobile engine or a heart pacemaker. Often, both *R* and *M* are used as the basis for the generation of tests and to determine the expected output as shown in Figure 5.7. In this chapter, we are concerned with the generation of tests using M.

The testing problem is to determine if the IUT is *equivalent* to M. For the FSM representation, equivalence was defined earlier. For other representations, equivalence is defined in terms of the input/output behavior of the IUT and M. As mentioned earlier, the protocol or design of interest might be implemented in hardware, software, or a combination of the two. Testing the IUT for equivalence against M requires executing it against a sequence of test inputs derived from M and comparing the observed behavior with the expected behavior as shown in Figure 5.7. Generation of tests from a formal representation of the design and their evaluation in terms of their fault detection ability is the key subject of this chapter.

5.4 A Fault Model

Given a set of requirements *R*, one often constructs a design from the requirements. An FSM is one possible representation of the design. Let M_d be a design intended to meet the requirements in *R*. Sometimes, M_d is referred to as a *specification* that guides the implementation. M_d is implemented in hardware or software. Let M_i denote an implementation that is intended to meet the requirements in *R* and has been derived

using M_d. Note that, in practice, M_i is unlikely to be an exact analog of M_d. In embedded real-time systems and communications protocols, it is often the case that M_i is a computer program that uses variables and operations not modeled in M_d. Thus, one could consider M_d as a finite state model of M_i.

The problem in testing is to determine whether or not M_i conforms to R. To do so, one tests M_i using a variety of inputs and checks for the correctness of the behavior of M_i with respect to R. The design M_d can be useful in generating a set T of tests for M_i. Tests so generated are also known as *black-box* tests, because they have been generated with reference to M_d and not M_i. Given T, one tests M_i against each test $t \in T$ and compares the behavior of M_i with the expected behavior given by exciting M_d in the initial state with test t.

Conformance testing, as presented in this chapter, is a black-box testing technique; tests are generated from the design and not the IUT.

In an ideal situation, one would like to ensure that any error in M_i is revealed by testing it against some test $t \in T$ derived from M_d. One reason why this is not feasible is that the number of possible implementations of a design M_d is infinite. This gives rise to the possibility of a large variety of errors one could introduce in M_i. In the face of this reality, a *fault model* has been proposed. This fault model defines a small set of possible fault types that can occur in M_i. Given a fault model, the goal is to generate a test set T from a design M_d. Any fault in M_i, of the type in the fault model, is guaranteed to be revealed when tested against T.

A widely used fault model for FSMs is shown in Figure 5.9. This figure illustrates four fault categories.

A fault model for testing using FSMs includes four types of errors: operation error, transfer error, extra state error, and missing state error. An operation error implies an incorrect output function. A transfer error implies an incorrect state transition function.

- *Operation error*: Any error in the output generated upon a transition is an operation error. This is illustrated by machines M and M_1 in Figure 5.9 where FSM M_1 generates an output 0, instead of a 1, when symbol a is input in state q_0. More formally, an operation error implies that for some state q_i in M and some input symbol s, $O(q_i, s)$ is incorrect.
- *Transfer error*: Any error in the transition from one state to the next is a transition error. This is illustrated by machine M_2 in Figure 5.9 where FSM M_2 moves to state q_1, instead of moving to q_0 from q_0 when symbol a is input. More formally, a transfer error implies that for some state q_i in M and some input symbol s, $\delta(q_i, x)$ is incorrect.
- *Extra state error*: An extra state may be introduced in the implementation. In Figure 5.9, machine M_3 has q_2 as the extra state when compared with the state set of machine M. However, an extra state may or may not be an error. For example, in Figure 5.10, machine M represents the correct design. Machines M_1 and M_2 have one extra state each. However, M_1 is an erroneous design but M_2 is not. This is because M_2 is actually equivalent to M even though it has an extra state.

Figure 5.9 Machine M represents the correct design. Machines M_1, M_2, M_3, and M_4 each contain an error.

A fault model aids in the assessment of the effectiveness of tests generated by techniques presented in this chapter.

Figure 5.10 Machine M represents the correct design. Machines M_1 and M_2 each have an extra state. However, M_1 is erroneous and M_2 is equivalent to M.

- *Missing state error*: A missing state is another type of error. We note from Figure 5.9 that machine M_4 has q_1 missing when compared with machine M. Given that the machine representing the correct design is minimal and complete, a missing state implies an error in the IUT.

The above fault model is generally used to evaluate a method for generating tests from a given FSM. The faults indicated above are also known collectively as *sequencing faults* or *sequencing errors*. It may be noted

that a given implementation could have more than one error of the same type. Other possibilities, such as two errors one each of a different type, also exist.

5.4.1 Mutants of FSMs

As mentioned earlier, given a design M_d, one could construct many correct and incorrect implementations. Let $\mathcal{I}(M_d)$ denote the set of all possible implementations of M_d. In order to make $\mathcal{I}$ finite, we assume that any implementation in $\mathcal{I}(M_d)$ differs from M_d in known ways. One way to distinguish an implementation from its specification is through the use of *mutants*. A mutant of M_d is an FSM obtained by introducing one or more errors one or more times. We assume that the errors introduced belong to the fault model described earlier. Using this method, we see in Figure 5.9 four mutants M_1, M_2, M_3, and M_4 of machine M. More complex mutants can also be obtained by introducing more than one error in a machine. Figure 5.11 shows two such mutants of machine M of Figure 5.9.

A slight variation, say M_1, of the correct design M is known as a mutant of M. Mutants are useful in assessing the adequacy of the generated tests. If M_1 is not equivalent to M then at least one test case, say t, must cause M_1 to behave differently than M in terms of the outputs generated. In such a situation M_1 is considered distinguished from M by test t.

Note that a mutant may be equivalent to M_d implying that the output behaviors of M_d and the mutant are identical on *all possible* inputs. Given a test set T, we say that a mutant is *distinguished* from M_d by a test $t \in T$, if the output sequence generated by the mutant is different from that generated by M_d when excited with t in their respective initial states. In this case, we also say that T distinguishes the mutant.

Using the idea of mutation, one can construct a finite set of possible implementations of a given specification M_d. Of course, this requires that some constraints are placed on the process of mutation, i.e. on how the errors are introduced. We caution the reader that the technique of testing software using program mutation is different from the use of mutation described here. This difference is discussed in Chapter 8. The next example illustrates how to obtain one set of possible implementations.

A first order mutant is obtain by making one change in an FSM. Higher order mutants are combinations of two or more first order mutants.

Example 5.6 Let M shown in Figure 5.12 denote the correct design. Suppose that we are allowed to introduce only one error at a time in M. This generates four mutants M_1, M_2, M_3, and M_4 shown in Figure 5.12 obtained by introducing an operation error. Considering that each state has two transitions, we get four additional mutants M_5, M_6, M_7, and M_8 by introducing the transfer error.

Introducing an additional state to create a mutant is more complex. First, we assume that only one extra state can be added to the FSM. However, there are several ways in which this state can interact with the existing states in the machine. This extra state

Figure 5.11 Two mutants of machine M. Mutant M_1 is obtained by introducing into M one operation error in state q_1 and one transfer error in state q_0 for input a. Mutant M_2 has been obtained by introducing an extra state q_2 in machine M and an operation error in state q_1 on input b.

Figure 5.12 Eight first-order mutants of M generated by introducing operation and transfer errors. Mutants M_1, M_2, M_3, and M_4 are generated by introducing operation errors in M. Mutants M_5, M_6, M_7, and M_8 are generated by introducing transfer errors in M.

can have two transitions that can be specified in 36 different ways depending on the tail state of the transition and the associated output function. We delegate to Exercise 5.10 the task of creating all the mutants of M by introducing an extra state.

Removing a state can be done in only two ways: remove q_0 or remove q_1. This generates two mutants. In general, this could be more complex when the number of states is greater than three. Removing a state will require the redirection of transitions and hence the number of mutants generated will be more than the number of states in the correct design.

Any non-empty test starting with symbol `a` distinguishes M from M_1. Mutant M_2 is distinguished from M by the input sequence `ab`. Note that when excited with `ab`, M outputs the string 11, whereas M_2 outputs the string 10. Input `ab` also distinguishes M_5 from M. The complete set of distinguishing inputs can be found using the W-method described in Section 5.6.

5.4.2 Fault coverage

One way to determine the "goodness" of a test set is to compute how many errors it reveals in a given implementation M_i. A test set that can reveal all errors in M_i is considered superior to one that fails to reveal one or more errors. Methods for the generation of test sets are often evaluated based on their fault coverage. The fault coverage of a test set is measured as a fraction between 0 and 1 and with respect to a given design specification. We give a formal definition of fault coverage in the following where the fault model presented earlier is assumed to generate the mutants.

Fault coverage is a measure of the goodness of a test set. It is defined as the ratio of the number of faults detected by a test set to the number of faults present in an IUT. Mutants can be used as a means to generate "faulty" designs.

N_t: Total number of first-order mutants of the machine M used for generating tests. This is also the same as $|\mathcal{I}(M)|$.

N_e: Number of mutants that are equivalent to M.

N_f: Number of mutants that are distinguished by test set T generated using some test generation method. Recall that these are the faulty mutants as no input in T is able to distinguish these mutants.

N_l: Number of mutants that are not distinguished by T.

The fault coverage of a test suite T with respect to a design M, and an implementation set $\mathcal{I}(M)$, is denoted by $FC(T, M)$ and computed as follows.

$$FC(T, M) = \frac{\text{Number of mutants not distinguished by } T}{\text{Number of mutants that are not equivalent to } M}$$
$$= \frac{N_t - N_e - N_f}{N_t - N_e}$$

In Section 5.9, we will see how FC can be used to evaluate the goodness of various methods to generate tests from FSMs. Next, we introduce the characterization set, and its computation, useful in various methods for generating tests from FSMs.

5.5 Characterization Set

The characterization set W for a minimal FSM M is a finite set of strings over the input alphabet of M such that for each pair of states (r, s) in M there exists at least one string in W that distinguishes state r from state s.

Most methods for generating tests from FSMs make use of an important set known as the *characterization set*. This set is usually denoted by W and is sometimes referred to as the W-set. In this section, we explain how one can derive a W-set, given the description of an FSM. Let us examine the definition of the W-set.

Let $M = (X, Y, Q, q_1, \delta, O)$ be an FSM that is minimal and complete. A characterization set for M, denoted as W, is a finite set of input sequences that distinguish the behavior of any pair of states in M. Thus, if q_i and q_j are states in Q, then there exists a string s in W such that $O(s, q_i) \neq O(s, q_j)$, where s belongs to X^+.

Example 5.7 Consider an FSM $M = (X, Y, Q, q_1, \delta, O)$ where $X = \{a, b\}$, $Y = \{0, 1\}$, and $Q = \{q_1, q_2, q_3, q_4, q_5\}$, q_1 is the initial state. The state transition function δ and the output function O are shown in Figure 5.13. The set W for this FSM is given below.

$$W = \{a, aa, aaa, baaa\}$$

A distinguishing sequence x for two states r and s of an FSM M causes M to produce different outputs when excited by x starting in states r or in s.

Let us do a sample check to determine if indeed, W is the characterization set for M. Consider the string `baaa`. It is easy to verify from Figure 5.13 that when M is started in state q_1 and excited with `baaa`, the output sequence generated is 1101. However, when M is started in state q_2 and excited with input `baaa`, the output sequence generated is 1100. Thus, we see that $O(\texttt{baaa}, q_1) \neq O(\texttt{baaa}, q_2)$ implying that the sequence `baaa` is a distinguishing sequence for states q_1 and q_2. You may now go ahead and perform similar checks for the remaining pairs of states.

Figure 5.13 The transition and output functions of a simple FSM.

The algorithm to construct a characterization set for an FSM M consists of two main steps. The first step is to construct a sequence of k-equivalence partitions $P_1, P_2, \ldots, P_m$ where $m > 0$. This iterative step converges in at most n steps where n is the number of states in M. In the second step, these k-equivalence partitions are traversed, in reverse order, to obtain the distinguishing sequences for every pair of states. These two steps are explained in detail in the following two subsections.

5.5.1 Construction of the *k*-equivalence partitions

Recall that given $M = (X, Y, Q, q_1, \delta, O)$, two states $q_i \in Q$ and $q_j \in Q$ are considered *k-equivalent*, if there does not exist an $s \in X^k$ such that $O(s, q_i) \neq O(s, q_j)$. The notion of *k-equivalence* leads to the notion of *k-equivalence partitions*.

A k-equivalence set consists of states that are k-equivalent, i.e., that cannot be distinguished by any string of length k or less over the input alphabet. A k-equivalent partition is a finite collection of k-equivalent sets such that each state in the FSM belongs to exactly one set. States that belong to different k-equivalent sets are considered k-distinguishable.

Given an FSM $M = (X, Y, Q, q_1, \delta, O)$, a *k-equivalence partition* of Q, denoted by P_k, is a collection of n finite sets of states denoted as $\Sigma_{k1}, \Sigma_{k2}, \ldots, \Sigma_{kn}$ such that

- $\bigcup_{i=1}^{n} \Sigma_{k_i} = Q$,
- States in Σ_{k_j}, for $1 \leq j \leq n$, are k-equivalent,
- If $q_l \in \Sigma_{k_i}$ and $q_m \in \Sigma_{k_j}$, for $i \neq j$, then q_l and q_m must be k-distinguishable.

We now illustrate the computation of k-equivalence partitions by computing them for the FSM of Example 5.7 using a rather long example.

Example 5.8 We begin by computing the 1-equivalence partition, i.e. P_1, for the FSM shown in Figure 5.13. To do so, write the transition and output functions for this FSM in a tabular form as shown below.

Current state	Output		Next state	
	a	b	a	b
q_1	0	1	q_1	q_4
q_2	0	1	q_1	q_5
q_3	0	1	q_5	q_1
q_4	1	1	q_3	q_4
q_5	1	1	q_2	q_5

State transition and output table for M.

The next step in constructing P_1 is to regroup the states so that all states that are identical in their output entries belong to the same group. Separation among two groups is indicated by a horizontal line as shown in the table below. Note from this table that states q_1, q_2, and q_3 belong to one group as they share identical output entries. Similarly, states q_4 and q_5 belong to another group. Notice also that we have added a Σ column to indicate group names. You may decide on any naming scheme for groups. In this example, groups are labeled as 1 and 2.

Σ	Current state	Output		Next state	
		a	b	a	b
1	q_1	0	1	q_1	q_4
	q_2	0	1	q_1	q_5
	q_3	0	1	q_5	q_1
2	q_4	1	1	q_3	q_4
	q_5	1	1	q_2	q_5

State transition and output table for M with grouping indicated.

The construction of P_1 is now complete. The groups separated by the horizontal line constitute a 1-equivalence partition. We have labeled these groups as 1 and 2. Thus, we get the 1-equivalence partition as

$P_1 = \{1, 2\}$

Group $1 = \Sigma_{11} = \{q_1, q_2, q_3\}$

Group $2 = \Sigma_{12} = \{q_4, q_5\}$

In preparation to begin the construction of the 2-equivalence partition, construct the P_1 table as follows. First, we copy the "Next State" sub-table. Next, rename each Next State entry by appending a second subscript which indicates the group to which that state belongs. For example, the next state entry under column "a" in the first row is q_1. As q_1 belongs to group 1, we relabel this as q_{11}. Other next state entries are relabeled in a similar way. This gives us the P_1 table as shown below.

Each P-table contains two or more groups of states. Each group of states is k-equivalent for some $k > 0$.

Σ	Current state	Next state	
		a	b
1	q_1	q_{11}	q_{42}
	q_2	q_{11}	q_{52}
	q_3	q_{52}	q_{11}
2	q_4	q_{31}	q_{42}
	q_5	q_{21}	q_{52}

P_1 table.

From the P_1 table given above, construct the P_2 table as follows. First, regroup all rows with identical *second* subscripts in its row entries under the Next State column. Note that the subscript we are referring to is the group label. Thus, for example, in q_{42} the subscript we refer to is 2 and not 4 2. This is because q_4 is the name of the state in machine M and 2 is the group label under the Σ column. As an example of regrouping in the P_1 table, the rows corresponding to the current states q_1 and q_2 have next states with identical subscripts and hence we group them together. This regrouping leads to additional groups. Next, relabel the groups and update the subscripts associated with the next state entries. Using this grouping scheme, we get the following P_2 table.

Σ	Current state	Next state	
		a	b
1	q_1	q_{11}	q_{43}
	q_2	q_{11}	q_{53}
2	q_3	q_{53}	q_{11}
3	q_4	q_{32}	q_{43}
	q_5	q_{21}	q_{53}

P_2 table.

Notice that there are three groups in the P_2 table. Regroup the entries in the P_2 table using the scheme described earlier for generating the P_2 table. This regrouping and relabeling gives us the following P_3 table.

Σ	Current state	Next state	
		a	b
1	q_1	q_{11}	q_{43}
	q_2	q_{11}	q_{54}
2	q_3	q_{54}	q_{11}
3	q_4	q_{32}	q_{43}
4	q_5	q_{21}	q_{54}

P_3 table.

Further regrouping and relabeling of P_3 table gives us the P_4 table given below.

Σ	Current state	Next state	
		a	b
1	q_1	q_{11}	q_{44}
2	q_2	q_{11}	q_{55}
3	q_3	q_{55}	q_{11}
4	q_4	q_{33}	q_{44}
5	q_5	q_{22}	q_{55}

P_4 table.

Note that no further partitioning is possible using the scheme described earlier. We have completed the construction of k-equivalence partitions, $k = 1, 2, 3, 4$, for machine M.

Notice that the process of deriving the P_k tables converged to P_4 and no further partitions are possible. See Exercise 5.6 that asks you to show that indeed the process will always converge. It is also interesting to note that each group in the P_4 table consists of exactly one state. This implies that all states in M are distinguishable. In other words, no two states in M are equivalent.

Let us now summarize the algorithm to construct a characterization set from the k-equivalence partitions and illustrate it using an example.

5.5.2 Deriving the characterization set

Having constructed the k-equivalence partitions, i.e. the P_k tables, we are now ready to derive the W-set for machine M. Recall that for each pair of states q_i and q_j in M, there exists at least one string in W that distinguishes q_i from q_j. Thus, the method to derive W proceeds by determining a distinguishing sequence for each pair of states in M. First, we sketch the method, referred to as the W procedure, and then illustrate it by a continuation of the previous example. In the procedure below, $G(q_i, x)$ denotes the label of the group to which an FSM moves when excited using input x in state q_i. For example, in the table for P_3, $G(q_2, b) = 4$ and $G(q_5, a) = 1$.

The characterization set is constructed from the P-tables by traversing the tables in the reverse order, i.e., the construction process begins with the last P-table and moves backwards.

The W-Procedure to derive W from a set of partition tables.

1. Let $M = (X, Y, Q, q_1, \delta, O)$ be the FSM for which $P = \{P_1, P_2, \ldots, P_n\}$ is the set of k-equivalence partition tables for $k = 1, 2, \ldots, n$. Initialize $W = \emptyset$.
2. Repeat the steps (a) through (d) for each pair of states (q_i, q_j), $i \neq j$, in M.

   a. Find an $r, 1 \leq r < n$ such that the states in pair (q_i, q_j) belong to the same group in P_r but not in P_{r+1}. In other words, P_r is the *last* of the P tables in which (q_i, q_j) belong to the same group. If such an r is found, then move to Step (b), otherwise find an $\eta \in X$ such that $O(q_i, \eta) \neq O(q_j, \eta)$, set $W = W \cup \{\eta\}$, and continue with the next available pair of states.
   The length of the minimal distinguishing sequence for (q_i, q_j) is $r + 1$. Denote this sequence as $z = x_0 x_1 \ldots x_r$ where $x_i \in X$ for $0 \leq i \leq r$.
   b. Initialize $z = \epsilon$. Let $p_1 = q_i$ and $p_2 = q_j$ be the *current* pair of states. Execute steps (i) through (iii) for $m = r, r-1, r-2, \ldots, 1$.

      (i) Find an input symbol η in P_m such that $G(p_1, \eta) \neq G(p_2, \eta)$. In case there is more than one symbol that satisfies the condition in this step, then select one arbitrarily.
      (ii) Set $z = z.\eta$.
      (iii) Set $p_1 = \delta(p_1, \eta)$ and $p_2 = \delta(p_2, \eta)$.

   c. Find an $\eta \in X$ such that $O(p_1, \eta) \neq O(p_2, \eta)$. Set $z = z.\eta$.
   d. The distinguishing sequence for the pair (q_i, q_j) is the sequence z. Set $W = W \cup \{z\}$.

Upon termination of the W procedure, we would have generated distinguishing sequences for all pairs of states in M. Note that the above algorithm is inefficient in that it derives a distinguishing sequence for each

pair of states even though two pairs of states might share the same distinguishing sequence. (See Exercise 5.7.) The next example applies the W procedure to obtain W for the FSM in Example 5.8.

Example 5.9 As there are several pairs of states in M, we illustrate how to find the distinguishing input sequences for the pairs (q_1, q_2) and (q_3, q_4). Let us begin with the pair (q_1, q_2).

According to Step 2(a) of the W procedure, we first determine r. To do so, note from Example 5.8 that the last P-table in which q_1 and q_2 appear in the same group is P_3. Thus $r = 3$.

Next, we move to Step 2(b) and set $z = \epsilon, p_1 = q_1, p_2 = q_2$. From P_3, we find that $G(p_1, b) \neq G(p_2, b)$ and update z to $z = zb$. Thus, b is the *first* symbol in the input string that distinguishes q_1 from q_2. In accordance with Step 2(b)(iii), reset $p_1 = \delta(p_1, b)$ and $p_2 = \delta(p_2, b)$. This gives us $p_1 = q_4$ and $p_2 = q_5$.

Going back to Step 2(b)(i), we find from the P_2 table that $G(p_1, a) \neq G(p_2, a)$. Update z to $z.a = ba$ and reset p_1 and p_2 to $p_1 = \delta(p_1, a)$ and $p_2 = \delta(p_2, a)$. We now have $p_1 = q_3$ and $p_2 = q_2$. Hence, a is the second symbol in the string that distinguishes states q_1 and q_2.

Once again return to Step 2(b)(i). We now focus our attention on the P_1 table and find that states $G(p_1, a) \neq G(p_2, a)$ and also $G(p_1, b) \neq G(p_2, b)$. We arbitrarily select a as the third symbol of the string that will distinguish states q_3 and q_2. In accordance with Steps 2(b)(ii) and 2(b)(iii), z, p_1, and p_2 are updated as $z = z.a = baa$, $p_1 = \delta(p_1, a)$ and $p_2 = \delta(p_2, a)$. This update leads to $p_1 = q_5$ and $p_2 = q_1$.

Finally at Step 2(c), the focus is on the original state transition table for M. States q_1 and q_5 in the table are distinguished by the symbol a. This is the last symbol in the string to distinguish states q_1 and q_2. Next, set $z = z.a = baaa$. The string $baaa$ is now discovered as the input sequence that distinguishes states q_1 and q_2. This sequence is added to W. Note from Example 5.7 that indeed baaa is a distinguishing sequence for states q_1 and q_2.

Next, the pair (q_3, q_4) is selected and the procedure resumes at Step 2(a). These two states are in a different group in table P_1 but in the same group in M. Thus, $r = 0$. As $O(q_3, a) \neq O(q_4, a)$, the distinguishing sequence for the pair (q_3, q_4) is the input sequence a.

It is left to the reader to derive distinguishing sequences for the remaining pairs of states. A complete set of distinguishing

Table 5.2 Distinguishing sequences for all pairs of states in the FSM of Example 5.8.

S_i	S_j	x	$o(S_i, x)$	$o(S_j, x)$
1	2	baaa	1	0
1	3	aa	0	1
1	4	a	0	1
1	5	a	0	1
2	3	aa	0	1
2	4	a	0	1
2	5	a	0	1
3	4	a	0	1
3	5	a	0	1
4	5	aaa	1	0

sequences for FSM M is given in Table 5.2. The leftmost two columns labeled S_i and S_j contain pairs of states to be distinguished. The column labeled x contains the distinguishing sequence for the pairs of states to its left. The rightmost two columns in this table show the *last* symbol output when input x is applied to state S_i and S_j, respectively. For example, as we have seen earlier $O(q_1, \texttt{baaa}) = 1101$. Thus, in Table 5.2, we only show the 1 in the corresponding output column. Notice that each pair in the last two columns is different, i.e. $O(S_i, x) \neq O(S_j, x)$ in each row. From this table, we obtain $W = \{a, aa, aaa, baaa\}$. While the sequence *aa* can be used instead of using *a*, doing so would lead to a longer tests.

5.5.3 Identification sets

Consider an FSM $M = (X, Y, Q, q_0, \delta, O)$ with symbols having their usual meanings and $|Q| = n$. Assume that M is completely specified and minimal. We know that a characterization set W for M is a set of strings such that for any pair of states q_i and q_j in M, there exists a string s in W such that $O(q_i, s) \neq (q_j, s)$.

For state s in a minimal FSM there exists an identification set. This set contains strings over the input alphabet that distinguish s from all other states in the FSM.

Analogous to the characterization set for M, we associate an *identification* set with each state of M. An identification set for state $q_i \in Q$ is

denoted by W_i and has the following properties: (a) $W_i \subseteq W$, $1 \leq i \leq n$, (b) $O(q_i, s) \neq O(q_j, s)$, for any j, $1 \leq j \leq n$, $j \neq i$, $s \in W_i$, and (c) no subset of W_i satisfies property (b). The next example illustrates the derivation of the identification sets given W.

The W- and the Wp-methods are both used for generating test sets from a minimal and complete FSM. The Wp-method will likely generate a smaller test set.

Example 5.10 Consider the machine given in Example 5.9 and its characterization set W shown in Table 5.2. From this table, we notice that state q_1 is distinguished from the remaining states by the strings `baaa`, `aa`, and `a`. Thus, $W_1 = \{baaa, aa, a\}$. Similarly, $W_2 = \{baaa, aa, a\}$, $W_3 = \{a, aa\}$, $W_4 = \{a, aaa\}$, and $W_5 = \{a, aaa\}$.

While the characterization sets are used in the W-method, the W_i sets are used in the Wp-method. The W- and the Wp-methods are used for generating tests from an FSM. We are now well equipped to describe various methods for test generation given an FSM. We begin with a description of the W-method.

5.6 The W-method

The W-method is used for constructing a test set from a given FSM M. The test set so constructed is a finite set of sequences that can be input to a program whose control structure is modeled by M. Alternately, the tests can also be the input to a design to test its correctness with respect to some specification.

The implementation modeled by M is also known as the Implementation Under Test and abbreviated as IUT. Note that most software systems cannot be modeled accurately using an FSM. However, the global control structure of a software system can be modeled by an FSM. This also implies that the tests generated using the W-method, or any other method based exclusively on a finite state model of an implementation, will likely reveal only certain types of faults. Later in Section 5.9, we discuss what kinds of faults are revealed by the tests generated using the W-method.

Generation of a test set from an FSM using the W-method requires that the FSM be minimal, complete, connected and deterministic. For the generated test s to be directly applicable to the IUT it is necessary that the input alphabets of the FSM and the IUT be the same.

5.6.1 Assumptions

The W-method makes the following assumptions for it to work effectively.

1. M is completely specified, minimal, connected, and deterministic.
2. M starts in a fixed initial state.
3. M can be reset accurately to the initial state. A `null` output is generated during the reset operation.
4. M and IUT have the same input alphabet.

Later in this section, we examine the impact of violating the above assumptions. Given an FSM $M = (X, Y, Q, q_0, \delta, O)$, the W-method consists of the following sequence of steps.

Step A Estimate the maximum number of states in the correct design.
Step B Construct the characterization set W for the given machine M.
Step C Construct the testing tree for M and determine the transition cover set P.
Step D Construct set Z.
Step E $P \cdot Z$ is the desired test set.

We already know how to construct W for a given FSM. In the remainder of this section, we explain each of the remaining three steps mentioned above.

5.6.2 Maximum number of states

We do not have access to the correct design or the correct implementation of the given specification. Hence, there is a need to estimate the maximum number of states in the correct design. In the worst case, i.e. when no information about the correct design or implementation is available, one might assume that the maximum number of states is the same as the number of states in M. The impact of an incorrect estimate is discussed after we have examined the W-method.

The number of states in the IUT might be more or less as compared to the number of states in the corresponding FSM. When no information is available about the IUT, one may assume that the number of states in the IUT is the same as that in the corresponding FSM.

5.6.3 Computation of the transition cover set

A transition cover set, denoted as P, is defined as follows. Let q_i and q_j, $i \neq j$, be two states in M. P consists of sequences sx such that $\delta(q_0, s) = q_i$ and $\delta(q_i, x) = q_j$. The empty sequence ϵ is also in P. We can construct P using the "testing tree" of M. A testing tree for an FSM is constructed as follows.

A transition cover set contains strings that when used to excite an FSM M ensure the coverage of all transitions in M. The set is constructed using a testing tree. The testing tree itself is constructed using the FSM.

1. State q_0, the initial state, is the root of the testing tree.
2. Suppose that the testing tree has been constructed until level k. The $(k+1)^{th}$ level is built as follows.

   - Select a node n at level k. If n appears at any level from 1 through k, then n is a leaf node and is not expanded any further. If n is not a leaf node then we expand it by adding a branch from node n to a new node m if $\delta(n, x) = m$ for $x \in X$. This branch is labeled as x. This step is repeated for all nodes at level k.

The following example illustrates construction of a testing tree for the machine in Example 5.8 whose transition and output functions are shown in Figure 5.13.

Example 5.11 The testing tree is initialized with the initial state q_1 as the root node. This is level 1 of the tree. Next, we note that $\delta(q_1, a) = q_1$ and $\delta(q_1, b) = q_4$. Hence, we create two nodes at the next level and label them as q_1 and q_4. The branches from q_1 to q_1 and q_4 are labeled, respectively, `a` and `b`. As q_1 is the only node at level 1, we now proceed to expand the tree from level 2.

At level 2, we first consider the node labeled q_1. However, another node labeled q_1 already appears at level 1, hence this node becomes a leaf node and is not expanded any further. Next, we examine the node labeled q_4. Note that $\delta(q_4, a) = q_3$ and $\delta(q_4, b) = q_4$. We therefore create two new nodes at level 3 and label these as q_4 and q_3 and label the corresponding branches as `b` and `a`, respectively.

We proceed in a similar manner down one level in each step. The method converges when at level 6 we have the nodes labeled q_1 and q_5 both of which appear at some level up the tree. We thus arrive at the testing tree for M as shown in Figure 5.14.

Once a testing tree has been constructed, the transition cover set P is obtained P by concatenating labels of all *partial paths* along the tree.

Figure 5.14 Testing tree for the machine with transition and output functions shown in Figure 5.13.

A partial path is a path starting from the root of the testing tree and terminating in any node of the tree. Traversing all partial paths in the tree shown in Figure 5.14, we obtain the following transition cover set.

A partial path in a testing tree is a path obtained by traversing the tree starting at the root and terminating at an internal or a leaf node.

$$P = \{\epsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\}$$

It is important to understand the function of the elements of P. As the name *transition cover set* implies, exciting an FSM in q_0, the initial state, with an element of P, forces the FSM into some state. After the FSM has been excited with all elements of P, each time starting in the initial state, the FSM has reached every state. Thus, exciting an FSM with elements of P ensures that all states are reached, and all transitions have been traversed at least once. For example, when the FSM M of Example 5.8 is excited by the input sequence *baab* in state q_1, it traverses the branches (q_1, q_4), (q_4, q_3), (q_3, q_5), and (q_5, q_5), in that order. The empty input sequence ϵ, does not traverse any branch but is useful in constructing the desired test sequence as is explained next.

Exciting an FSM once with each element of the transition cover set ensures coverage of all its states and transitions.

5.6.4 Constructing *Z*

Given the input alphabet X and the characterization set W, it is straightforward to construct Z. Suppose that the number of states estimated to be in the IUT is m and the number of states in the design specification is n, $m \geq n$. Given this information, Z can be computed as follows.

$$Z = (X^0 \cdot W) \cup (X \cdot W) \cup (X^2 \cdot W) \ldots \cup (X^{m-1-n} \cdot W) \cup (X^{m-n} \cdot W)$$

It is easy to see that $Z = X \cdot W$ for $m = n$, i.e. when the number of states in the IUT is the same as that in the specification. For $m < n$, we use $Z = X \cdot W$. Recall that $X^0 = \{\epsilon\}$, $X^1 = X$, $X^2 = X \cdot X$, and so on, where the symbol "·" denotes the string concatenation operation. For convenience, we shall use the shorthand notation $X[p]$ to denote the following set union

$$\{\epsilon\} \cup X^1 \cup X^2 \ldots \cup X^{p-1} \cup X^p.$$

We can now rewrite Z as $Z = X[m-n] \cdot W$, for $m \geq n$ where m is the number of states in the IUT and n the number of states in the design specification.

5.6.5 Deriving a test set

Having constructed P and Z, we can easily obtain a test set T as $P \cdot Z$. The next example shows how to construct a test set from the FSM of Example 5.8.

Example 5.12 For the sake of simplicity, let us assume that the number of states in the correct design or IUT is the same as in the

given design shown in Figure 5.13. Thus, $m = n = 5$. This leads to the following Z:

$Z = X^0 \cdot W = \{a, aa, aaa, baaa\}$

Catenating P with Z, we obtain the desired test set.

$$
\begin{aligned}
T = P \cdot Z = \{&\epsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\} \cdot \{a, aa, \\
&aaa, baaa\} \\
= \{&a, aa, aaa, baaa, \\
&aa, aaa, aaaa, abaaa, \\
&ba, baa, baaa, bbaaa, \\
&bba, bbaa, bbaaa, bbbaaa, \\
&baa, baaa, baaaa, babaaa, \\
&baba, babaa, babaaa, babbaaa, \\
&baaa, baaaa, baaaaa, baabaaa, \\
&baaba, baabaa, baabaaa, baabbaaa, \\
&baaaa, baaaaa, baaaaaa, baaabaaa \\
&baaaba, baaabaa, baaabaaa, baaabbaaa \\
&baaaaa, baaaaaa, baaaaaaa, baaaabaaa\}
\end{aligned}
$$

Assuming that the IUT has one extra state, i.e. $m = 6$, we obtain Z and the test set $P \cdot Z$ as follows.

$$
\begin{aligned}
Z = X^0 \cdot W \cup X^1 \cdot W = \{&a, aa, aaa, baaa, aa, aaa, aaaa, abaaa, \\
&ba, baa, baaa, bbaaa\}
\end{aligned}
$$

$$
\begin{aligned}
T = P \cdot Z = \{&\epsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\} \cdot \\
&\{a, aa, aaa, baaa, aa, aaa, aaaa, abaaa, ba, baa, baaa, bbaaa\}
\end{aligned}
$$

5.6.6 Testing using the W-method

To test a given IUT M_i against its specification M, we do the following for each test input.

1. Find the expected response $M(t)$ to a given test input t. This is done by examining the specification. Alternately, if a tool is available, and the

specification is executable, one could determine the expected response automatically.

2. Obtain the response $M_i(t)$ of the IUT, when excited with t in the initial state.
3. If $M(t) = M_i(t)$, then no flaw has been detected so far in the IUT. $M(t) \neq M_i(t)$ implies the possibility of a flaw in the IUT under test, given a correct design.

Notice that a mismatch between the expected and the actual response does not necessarily imply an error in the IUT. However, if we assume that (a) $M(t)$ and $M_i(t)$ have been determined without any error, and (b) the comparison between $M(t)$ and $M_i(t)$ is correct, then indeed $M(t) \neq M_i(t)$ implies an error in the design or the IUT.

A mismatch between the expected output derived from an FSM and the actual output obtained from the IUT does not necessarily imply an error in the IUT.

Example 5.13 Let us now apply the W-method to test the design shown in Figure 5.13. We assume that the specification is also given in terms of an FSM which we refer to as the "correct design." Note that this might not happen in practice but we make this assumption for illustration.

Consider two possible implementations under test. The correct design is shown in Figure 5.15(a) and is referred to as M. We denote the two erroneous designs, corresponding to two IUTs under test, as M_1 and M_2, respectively, shown in Figure 5.15(b) and (c).

Notice that M_1 has one transfer error with respect to M. The error occurs in state q_2 where the state transition on input `a` should be $\delta_1(q_2, a) = q_1$ and not $\delta_1(q_2, a) = q_3$. M_2 has two errors with respect to M. There is a transfer error in state q_2 as mentioned earlier. In addition, there is an operation error in state q_5 where the output function on input `b` should be $O_2(q_5, b) = 1$ and not $O_2(q_5, b) = 0$.

To test M_1 against M, we apply each input t from the set $P \cdot Z$ derived in Example 5.12 and compare $M(t)$ with $M_1(t)$. However, for the purpose of this example, let us first select $t = ba$. Tracing gives us $M(t) = 11$ and $M_1(t) = 11$. Thus, IUT M_1 behaves correctly when excited by the input sequence *ba*. Next, we select $t = baaaaaa$ as a test input. Tracing the response of $M(t)$ and $M_1(t)$ when excited by t, we obtain $M(t) = 1101000$ and $M_1(t) = 1101001$. Thus, the input sequence *baaaaaa* has revealed the transfer error in M_1.

Next, let us test M_2 with respect to specification M. We select the input sequence $t = baaba$ for this test. Tracing the two designs we obtain $M(t) = 11011$, whereas $M_2(t) = 11001$. As the two traces are different, input sequence *baaba* reveals the operation error. We have

Figure 5.15 The transition and output functions of FSMs of the design under test, denoted as M in the text and copied from Figure 5.13 and two incorrect designs in (b) and (c) denoted as M_1 and M_2 in the text.

Tests generated from an FSM are assumed independent and need to be applied to the IUT in the start state. Thus, prior to applying a test input, the IUT needs to be brought to its start state.

already shown that $x = baaaaaa$ reveals the transfer error. Thus, the two input sequences *baaba* and *baaaaaa* in $P \cdot Z$ reveal the errors in the two IUTs under test.

Note that for the sake of brevity, we have used only two test inputs. However, in practice, one needs to apply test inputs to the IUT in some sequence until the IUT fails or the IUT has performed successfully on all tests.

Example 5.14 Suppose it is estimated that the IUT corresponding to the machine in Figure 5.13 has one extra state, i.e. $m = 6$. Let us also assume that the IUT, denoted as M_1, is as shown in Figure 5.16(a) and indeed has six states. Testing M_1 against $t = baaba$ leads to $M_1(t) = 11001$. The correct output obtained by tracing M against t is 11011. As $M(t) \neq M_1(t)$, test input t has revealed the "extra state" error in S_1.

Figure 5.16 Two implementations each of the design in Figure 5.13 each having an extra state. Note that the output functions of the extra state q_6 are different in (a) and (b)

However, test $t = baaba$ does not reveal the extra state error in machine M_2 as $M_2(t) = 11011 = M(t)$. Consider now test $t = baaa$. We get $M(t) = 1101$ and $M_2(t) = 1100$. Thus, the input *baaa* reveals the extra state error in machine M_2.

5.6.7 The error detection process

Each test derived using the W-method is of the kind *rs* where *r* is derived from the testing tree and *s* is an element of the characterization set.

Let us now carefully examine how the test sequences generated by the W-method detect operation and transfer errors. Recall that the test set generated in the W-method is the set $P \cdot W$, given that the number of states in the IUT is the same as that in the specification. Thus, each test case t is of the form $r \cdot s$ where r belongs to the transition cover set P and s to the characterization set W. We consider the application of t to the IUT as a two-phase process. In the first phase, input r moves the IUT from its initial state q_0 to some state q_i. In the second phase, the remaining input s moves the IUT to its final state q_j or $q_{j'}$. These two phases are shown in Figure 5.17.

When the IUT is started in its initial state, say q_0, and excited with test t, it consumes the string r and arrives at some state q_i as shown in Figure 5.17. The output generated so far by the IUT is u where $u = O(q_0, r)$. Continuing further in state q_i, the IUT consumes symbols in string s and arrives at state q_j. The output generated by the IUT while moving from state q_i to q_j is v where $v = O(q_i, s)$. If there is any operation error along the transitions traversed between states q_0 and q_i, then the output string u would be different from the output generated by the specification machine M when excited

Figure 5.17 Detecting errors using tests generated by the W-method.

with r in its initial state. If there is any operation error along the transitions while moving from q_i to q_j, then the output wv would be different from that generated by M.

The detection of a transfer error is more involved. Suppose there is a transfer error in state q_i and $s = as'$. Thus, instead of moving to state q_k, where $q_k = \delta(q_i, a)$, the IUT moves to state $q_{k'}$ where $q_{k'} = \delta(q_i, a)$. Eventually, the IUT ends up in state $q_{j'}$ where $q_{j'} = \delta(q_{k'}, s')$. Note that this step may take several transitions and hence is indicated by a dotted arrow in the figure. Our assumption is that s is a distinguishing sequence in W for states q_i and $q_{j'}$. In this case, $wv' \neq wv$. If s is not a distinguishing sequence for q_i and $q_{j'}$ then there must exist some other input as'' in W such that $wv'' \neq wv$ where $v'' = O(q_{k'}, s'')$.

5.7 The Partial W-method

The partial W-method, referred to as the Wp-method, uses the state cover set and the state identification sets. Tests so generated are generally shorter than those generated by the W-method.

The partial W-method, also known as the *Wp-method*, is similar to the W-method in that tests are generated from a minimal, complete, and connected FSM. However, the size of the test set generated using the Wp-method is often smaller than that generated using the W-method. This reduction is achieved by dividing the test generation process into two phases and making use of the state identification sets W_i, instead of the characterization set W, in the second phase. Furthermore, the fault detection effectiveness of the Wp-method remains the same as that of the W-method. The two-phases used in the Wp-method are described later in Section 5.7.1.

First, we define a *state cover set* S of an FSM $M = (X, Y, Q, q_0, \delta, O)$. S is a finite non-empty set of sequences where each sequence belongs to X^*

and for each state $q_i \in Q$, there exists an $r \in S$ such that $\delta(q_0, r) = q_i$. It is easy to see that the state cover set is a subset of the transition cover set and is not unique. Note that the empty string ϵ belongs to S which *covers* the initial state because, as per the definition of state transitions, $\delta(q_0, \epsilon) = q_0$.

For each state s in an FSM a state cover set contains at least one element that moves the FSM from its initial state to s. The empty string is a part of the state cover set.

Example 5.15 In Example 5.11 the following transition cover set was constructed for machine M of Figure 5.13.

$P = \{\epsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\}$

The following subset of P forms a state cover set for M.

$S = \{\epsilon, b, ba, baa, baaa\}$

We are now ready to show how tests are generated using the Wp-method. As before, let M denote the FSM representation of the specification against which we are to test an IUT. Suppose that M contains $n > 0$ states and the IUT contains $m > 0$ states. The test set T derived using the Wp-method is composed of subsets T_1 and T_2, i.e. $T = T_1 \cup T_2$. Computation of subsets T_1 and T_2 is shown below assuming that M and the IUT have an equal number of states, i.e. $m = n$.

```
Procedure for test generation using the Wp-method.
```

Step 1 Compute the transition cover set P, the state cover set S, the characterization set W, and the state identification sets W_i for M. Recall that S can be derived from P and the state identification sets can be computed as explained in Example 5.10.

Step 2 $T_1 = S \cdot W$.

Step 3 Let $\mathcal{W}$ be the set of all state identification sets of M, i.e. $\mathcal{W} = \{W_1, W_2, \ldots, W_n\}$.

Step 4 Let $R = \{r_{i_1}, r_{i_2}, \ldots r_{i_k}\}$ denote the set of all sequences that are in the transition cover set P but not in the state cover set S, i.e. $R = P - S$. Furthermore, let $r_{i_j} \in R$ be such that $\delta(q_0, r_{i_j}) = q_{i_j}$.

Step 5 $T_2 = R \bigotimes \mathcal{W} = \bigcup_{j=1}^{k} (\{r_{i_j}\} \cdot W_{i_j})$, where $W_{i_j} \in \mathcal{W}$ is the state identification set for state q_{i_j}.

```
End of Procedure for test generation using the
Wp-method.
```

The $\bigotimes$ operator used in the derivation of T_2 is known as the *partial string concatenation operator*. Having constructed the test set T, which consists of subsets T_1 and T_2, one proceeds to test the IUT as described in the next section.

The partial string concatenation operator is used in the construction of a subset of tests when using the Wp-method.

5.7.1 Testing using the Wp-method for $m = n$

Given a specification machine M and its implementation under test, the Wp-method consists of a two-phase test procedure. In the first phase, the IUT is tested using the elements of T_1. In the second phase, the IUT is tested against the elements of T_2. Details of the two phases follow.

Phase 1: Testing the IUT against each element of T_1 tests each state for equivalence with the corresponding state in M. Note that a test sequence t in T_1 is of the kind uv where $u \in S$ and $v \in W$. Suppose that $\delta(q_0, u) = q_i$ for some state q_i in M. Thus, the application of t first moves M to some state q_i. Continuing with the application of t to M, now in state q_i, generates an output different from that generated if v were applied to M in some other state q_j, i.e. $O(q_i, v) \neq O(q_j, v)$, for $i \neq j$. If the IUT behaves as M against each element of T_1, then the two are equivalent with respect to $S \cdot W$. If not, then there is an error in the IUT.

Phase 2: While testing the IUT using elements of T_1 checks all states in the implementation against those in M, it may miss checking all transitions in the IUT. This is because T_1 is derived using the state cover set S and not the transition cover set P. Elements of T_2 complete the checking of the remaining transitions in the IUT against M.

To understand how, we note that each element t of T_2 is of the kind uv where u is in P but not in S. Thus, the application of t to M moves it to some state q_i where $\delta(q_0, u) = q_i$. However, the sequence of transitions that M goes through is different from the sequence it goes through while being tested against elements of T_1 because $u \notin S$. Next, M which is now in state q_i, is excited with v. Now v belongs to W_i which is the state identification set of q_i and hence this portion of the test will distinguish q_i from all other states with respect to transitions not traversed while testing against T_1. Thus, any transfer or operation errors in the transitions not tested using T_1 will be discovered using T_2.

Example 5.16 Let us reconsider the machine in Figure 5.13 to be the specification machine M. First, various sets needed to generate the test set T are reproduced below for convenience.

$$
\begin{aligned}
W &= \{a, aa, aaa, baaa\} \\
P &= \{\epsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\} \\
S &= \{\epsilon, b, ba, baa, baaa\} \\
W_1 &= \{baaa, aa, a\} \\
W_2 &= \{baaa, aa, a\} \\
W_3 &= \{a, aa\} \\
W_4 &= \{a, aaa\} \\
W_5 &= \{a, aaa\}
\end{aligned}
$$

Next, we compute set T_1 using Step 2.

$$\begin{aligned} T_1 = S \cdot W = \quad & \{a, aa, aaa, baaa, \\ & ba, baa, baaa, bbaaa, \\ & baa, baaa, baaaa, babaaa \\ & baaa, baaaa, baaaaa, baabaaa, \\ & baaaa, baaaaa, baaaaaa, baaabaaa\} \end{aligned}$$

Next, in accordance with Step 4 we obtain R as

$R = P - S = \{a, bb, bab, baab, baaab, baaaa\}$.

With reference to Step 4 and Figure 5.13, we obtain the following state transitions for the elements in R:

$$\begin{aligned} \delta(q_1, a) &= q_1 \\ \delta(q_1, bb) &= q_4 \\ \delta(q_1, bab) &= q_1 \\ \delta(q_1, baab) &= q_5 \\ \delta(q_1, baaab) &= q_5 \\ \delta(q_1, baaaa) &= q_1 \end{aligned}$$

From the transitions given above, we note that when M is excited by elements a, bb, bab, $baab$, $baaab$, and $baaaa$, starting on each occasion in state q_1, it moves to states, q_1, q_4, q_1, q_5, q_5, and q_1, respectively. Thus, while computing T_2, we need to consider the state identification sets W_1, W_4, and W_5. Using the formula in Step 5, we obtain T_2 as follows.

$$\begin{aligned} T_2 = R \bigotimes \mathcal{W} &= (\{a\} \cdot W_1) \cup (\{bb\} \cdot W_4) \cup (\{bab\} \cdot W_5) \cup (\{baab\} \cdot W_5) \cup \\ & \quad (\{baaab\} \cdot W_5) \cup (\{baaaa\} \cdot W_1) \\ &= \{abaaa, aaa, aa\} \cup \{bba, bbaaa\} \cup \{baba, babaaa\} \cup \\ & \quad \{baaba, baabaaa\} \cup \{baaaba, baaabaaa\} \cup \\ & \quad \{baaaabaaa, baaaaaa, baaaaa\} \\ &= \{abaaa, aaa, aa, bba, bbaaa, baba, babaaa, baaba, \\ & \quad baabaaa, baaaba, baaabaaa, baaaabaaa, \\ & \quad baaaaaa, baaaaa\} \end{aligned}$$

The desired test set $T = T_1 \cup T_2$. Note that T contains a total of 21 tests. This is in contrast to the 29 tests generated using the W-method in Example 5.12 when $m = n$.

The next example illustrates the Wp-method.

Example 5.17 Let us assume we are given the specification M as shown in Figure 5.15(a) and are required to test IUTs that correspond to designs M_1 and M_2 in Figure 5.15(b) and (c), respectively. For each test, we proceed in two phases as required by the Wp-method.

Testing using the Wp-method proceeds in two phases. In the first phase, elements of test T_1 are applied followed by those of test T_2 where T_1 and T_2 are subsets of the tests generated using the Wp-method. Tests in T_1 ensure that each state is reached and distinguished from the rest. However, doing so might miss the testing of all transitions which is done in phase 2 using tests from T_2.

Test M_1, phase 1: We apply each element t of T_1 to the IUT corresponding to M_1 and compare $M_1(t)$ with the expected response $M(t)$. To keep this example short, let us consider test $t = baaaaaa$. We obtain $M_1(t) = 1101001$. On the contrary, the expected response is $M(t) = 1101000$. Thus, test input t has revealed the transfer error in state q_2.

Test M_1, phase 2: This phase is not needed to test M_1. In practice, however, one would test M_1 against all elements of T_2. You may verify that none of the elements of T_2 reveal the transfer error in M_1.

Test M_2, phase 1: Test $t = baabaaa$ that belongs to T_1 reveals the error as $M_2(t) = 1100100$ and $M(t) = 1101000$.

Test M_2, phase 2: Once again this phase is not needed to reveal the error in M_2.

Note that in both the cases in the example above, we do not need to apply phase 2 of the Wp-method to reveal the error. For an example that does require phase 2 to reveal an implementation error, refer to Exercise 5.15 that shows an FSM and its erroneous implementation and requires the application of phase 2 for the error to be revealed. However, in practice, one would not know whether or not phase 1 is sufficient. This would lead to the application of both phases and hence all tests. Note that tests in phase 2 ensure the coverage of all transitions. While tests in phase 1 might cover all transitions, they might not apply the state identification inputs. Hence, all errors corresponding to the fault model in Section 5.4 are not guaranteed to be revealed by tests in phase 1.

5.7.2 Testing using the Wp-method for $m > n$

The procedure for constructing tests using the Wp-method can be modified easily when the number of states in the IUT is estimated to be larger than that in the specification, i.e. when $m > n$. Steps 2 and 5 of the procedure described earlier need to be modified.

The Wp-method described earlier needs modification when the IUT has more states than the FSM from which the tests are generated.

For $m = n$, we compute $T_1 = S \cdot W$. For $m > n$ we modify this formula so that $T_1 = S \cdot Z$ where $Z = X[m-n] \cdot W$ as explained in Section 5.6.4. Recall that T_1 is used in phase 1 of the test procedure. T_1 is different from T derived from the W-method in that it uses the state cover set S and not the transition cover set P. Hence, T_1 contains fewer tests than T except when $P = S$.

To compute T_2, we first compute R as in Step 4. Recall that R contains only those elements of the transition cover set P that are not in the state cover set S. Let $R = P - S = \{r_{i_1}, r_{i_2}, \ldots r_{i_k}\}$. As before, $r_{i_j} \in R$ moves M

from its initial state to state q_{i_j}, i.e. $\delta(q_0, r_{i_j}) = q_{i_j}$. Given R, we derive T_2 as follows.

$$T_2 = R \cdot X[m-n] \bigotimes \mathcal{W} = \bigcup_{j=1}^{k} \{r_{i_j}\} \cdot (\bigcup_{u \in X[m-n]} u \cdot W_l),$$

where $\delta(q_0, r_{i_j}) = q_{i_j}$, $\delta(q_{i_j}, u) = q_l$, and $W_l \in \mathcal{W}$ is the identification set for state q_l.

The basic idea underlying phase 2 is explained in the following. The IUT under test is exercised so that it reaches some state q_i from the initial state q_0. Let u denote the sequence of input symbols that move the IUT from state q_0 to q_i. As the IUT contains $m > n$ states, it is now forced to take an additional $(m-n)$ transitions. Of course, it requires $(m-n)$ input symbols to force the IUT for that many transitions. Let v denote the sequence of symbols that move the IUT from state q_i to state q_j in $(m-n)$ steps. Finally, the IUT receives an input sequence of symbols, say w, that belong to the state identification set W_j. Thus, one test for the IUT in phase 2 is comprised of the input sequence uvw.

Notice that the expression to compute T_2 consists of two parts. The first part is R and the second part is the partial string concatenation of $X[m-n]$ and $\mathcal{W}$. Thus, a test in T_2 can be written as uvw where $u \in R$ is the string that takes the IUT from its initial state to some state q_i, string $v \in X[m-n]$ takes the IUT further to state q_j and string $w \in W_j$ takes it to some state q_l. If there is no error in the IUT, then the output string generated by the IUT upon receiving the input uvw must be the same as the one generated when the design specification M is exercised by the same string.

Example 5.18 Let us construct a test set using the Wp-method for machine M in Figure 5.13 given that the corresponding IUT contains an extra state. For this scenario, we have $n = 5$ and $m = 6$. Various sets needed to derive T are reproduced here for convenience.

$$\begin{aligned}
X &= \{a, b\} \\
W &= \{a, aa, aaa, baaa\} \\
P &= \{\epsilon, a, b, bb, ba, baa, bab, baab, baaa, baaab, baaaa\} \\
S &= \{\epsilon, b, ba, baa, baaa\} \\
W_1 &= \{baaa, aa, a\} \\
W_2 &= \{baaa, aa, a\} \\
W_3 &= \{a, aa\} \\
W_4 &= \{a, aaa\} \\
W_5 &= \{a, aaa\}
\end{aligned}$$

First, T_1 is derived as $S \cdot X[1] \cdot W$. Recall that $X[1]$ denotes the set $\{\epsilon\} \cup X^1$.

$$
\begin{aligned}
T_1 &= S \cdot (\{\epsilon\} \cup X^1) \cdot W \\
&= (S \cdot W) \cup (S \cdot X \cdot W) \\
S \cdot W &= \{a, aa, aaa, baaa, \\
&\quad ba, baa, baaa, bbaaa, \\
&\quad baa, baaa, baaaa, babaaa, \\
&\quad baaa, baaaa, baaaaa, baabaaa, \\
&\quad baaaa, baaaaa, baaaaaa, baaabaaa\} \\
S \cdot X &= \{a, b, ba, bb, baa, bab, baaa, baab, baaaa, baaab\} \\
S \cdot X \cdot W &= \{aa, aaa, aaaa, abaaa, \\
&\quad ba, baa, baaa, bbaaa, \\
&\quad baa, baaa, baaaa, babaaa, \\
&\quad bba, bbaa, bbaaa, bbbaaa, \\
&\quad baaa, baaaa, baaaaa, baabaaa, \\
&\quad baba, babaa, babaaa, babbaaa, \\
&\quad baaaa, baaaaa, baaaaaa, baaabaaa, \\
&\quad baaba, baabaa, baabaaa, baabbaaa, \\
&\quad baaaaa, baaaaaa, baaaaaaa, baaaabaaa, \\
&\quad baaaba, baaabaa, baaabaaa, baaabbaaa\}
\end{aligned}
$$

T_1 contains a total of 60 tests of which 20 are in $S \cdot W$ and 40 in $S \cdot X \cdot W$. To obtain T_2, we note that $R = P - S = \{a, bb, bab, baab, baaab, baaaa\}$. T_2 can now be computed as follows.

$$
\begin{aligned}
T_2 &= R \cdot X[m-n] \bigotimes \mathcal{W} \\
&= R \cdot X[1]) \bigotimes \mathcal{W} \\
R \bigotimes \mathcal{W} &= (\{a\} \cdot W_1) \cup (\{bb\} \cdot W_4) \cup (\{baab\} \cdot W_5) \cup (\{bab\} \cdot W_1) \cup \\
&\quad (\{baaab\} \cdot W_1) \cup (\{baaaa\} \cdot W_5) \\
&= \{abaaa, aaa, aa, bba, bbaaa, babbaaa, babaa, baba, \\
&\quad baaba, baabaaa, baaaba, baaabaaa, baaaabaaa, \\
&\quad baaaaaa, baaaaa\} \\
(R \cdot X) \bigotimes \mathcal{W} &= \{aa, ab, bba, bbb, baba, babb, baaba, baabb, baaaba, \\
&\quad baaabb, baaaaa, baaaab\} \bigotimes \mathcal{W} \\
&= (aa \cdot W_1) \cup (ab \cdot W_4) \cup \\
&\quad (bba \cdot W_3) \cup (bbb \cdot W_4) \cup \\
&\quad (baaba \cdot W_2) \cup (baabb \cdot W_5) \\
&\quad (baba \cdot W_1) \cup (babb \cdot W_4) \\
&\quad (baaaba \cdot W_2) \cup (baaabb \cdot W_5) \\
&\quad (baaaaa \cdot W_1) \cup (baaaab \cdot W_5) \\
&= \{aabaaa, aaaa, aaa, aba, abaaa, bbaa, bbaaa, bbba, \\
&\quad bbbaaa, bababaaa, babaaa, babaa, babba,
\end{aligned}
$$

babbaaa, *baababaaa*, *baabaaa*, *baabaa*, *baabba*,
baabbaaa, *baaababaaa*, *baaabaaa*, *baaabaa*,
baaabba, *baaabbaaa*, *baaaaabaaa*, *baaaaaaa*,
baaaaaa, *baaaaba*, *baaaabaaa*}

T_2 contains a total of 38 tests. In all the entire test set $T = T_1 \cup T_2$ contains 81 tests. This is in contrast to a total of 44 tests that would be generated by the W-method. (See Exercises 5.16 and 5.17.)

5.8 The UIO-Sequence Method

The W-method uses the characterization set W of distinguishing sequences as the basis to generate a test set for an IUT. The tests so generated are effective in revealing operation and transfer faults. However, the number of tests generated using the W-method is usually large. Several alternative methods have been proposed that generate fewer test cases. In addition, these methods are either as effective or nearly as effective as the W-method in their fault detection capability. We now introduce two such methods. A method for test generation based on *unique input/output sequences* is presented in this section.

5.8.1 Assumptions

The UIO method generates tests from an FSM representation of the design. All the assumptions stated in Section 5.6.1 apply to the FSM that is input to the UIO test generation procedure. In addition, it is assumed that the IUT has the same number of states as the FSM that specifies the corresponding design. Thus, any errors in the IUT are transfer and operations errors only as shown in Figure 5.12.

5.8.2 UIO sequences

A Unique Input/Output (UIO) sequence is a sequence of input/output pairs that are unique for each state in an FSM. Such a sequence might not exist.

A unique input/output sequence, abbreviated as UIO sequence, is a sequence of input and output pairs that distinguishes a state of an FSM from the remaining states. Consider FSM $M = (X, Y, Q, q_0, \delta, O)$. A UIO sequence of length k for some state $s \in Q$ is denoted as $UIO(s)$ and looks like the following sequence

$$UIO(s) = i_1/o_1 \cdot i_2/o_2 \cdot \ldots i_{(k-1)}/o_{(k-1)} \cdot i_k/o_k.$$

In the sequence given above, each pair a/b consists of an input symbol a that belongs to the input alphabet X and an output symbol b

that belongs to the output alphabet Y. As before, the dot symbol ($\cdot$) denotes string concatenation. We refer to the *input* portion of a $UIO(s)$ as $in(UIO(s))$ and to its *output* portion as $out(UIO(s))$. Using this notation for the input and output portions of $UIO(s)$, we can rewrite $UIO(s)$ as

$in(UIO(s)) = i_1 \cdot i_2 \cdot \ldots i_{(k-1)} \cdot i_k$, and $out(UIO(s)) = o_1 \cdot o_2 \cdot \ldots o_{(k-1)} \cdot o_k$.

When the sequence of input symbols that belong to $in(UIO(s))$ is applied to M in state s, the machine moves to some state t and generates the output sequence $out(UIO(s))$. This can be stated more precisely as follows,

$\delta(s, in(UIO(s)) = t,\ O(s, in(UIO(s))) = out(UIO(s))$.

There is one UIO sequence for each state s in an FSM. The existence of such a sequence is not guaranteed.

A formal definition of unique input/output sequences follows. *Given an FSM $M = (X, Y, Q, q_0, \delta, O)$, the unique input/output sequence for state $s \in Q$, denoted as $UIO(s)$, is a sequence of one or more edge labels such that the following condition holds.*

$O(s, in(UIO(s))) \neq O(t, in(UIO(s)))$, for all $t \in Q, t \neq s$.

The next example offers UIO sequences for a few FSMs. The example also shows that there may not exist any UIO sequence for one or more states in an FSM.

Example 5.19 Consider machine M_1 shown in Figure 5.18. The UIO sequence for each of the six states in M_1 is given below.

Using the definition of UIO sequences, it is easy to check whether or not a $UIO(s)$ for some state s is indeed a unique input/output sequence. However, before we perform such a check,

Figure 5.18 Two FSMs used in Example 5.19. There is a UIO sequence for each state in Machine M_1. There is no UIO sequence for state q_1 in Machine M_2.

State (s)	UIO (s)
q_1	a/0 · c/1
q_2	c/1· c/1
q_3	b/1· b/1
q_4	b/1· c/0
q_5	c/0
q_6	c/1· a/0

we assume that the machine generates an empty string as the output if there does not exist an outgoing edge from a state on some input. For example, in Machine M_1 there is no outgoing edge from state q_1 for input c. Thus, M_1 generates an empty string when it encounters a c in state q_1. This behavior is explained in Section 5.8.3.

Let us now perform two sample checks. From the table above we have $UIO(q_1) = a/0 \cdot c/1$. Thus, $in(UIO(q_1) = ac$, $out(UIO(q_1)) = 01$. Applying the sequence $a \cdot c$ to state q_2 produces the output sequence 0 which is different from 0_1 generated when $a \cdot c$ is applied to M1 in state q_1. Similarly, applying ac to Machine M_1 in state q_5 generates the output pattern 0 which is, as before, different from that generated by q_1. You may now perform all other checks to ensure that indeed the UIOs given above are correct.

Example 5.20 Now consider Machine M_2 in Figure 5.18. The UIO sequences for all states, except state q_1, are listed below. Notice that there does not exist any UIO sequence for state q_1.

State (s)	UIO (s)
q_1	None
q_2	a/1
q_3	b/1

5.8.3 Core and non-core behavior

An FSM must be connected for the UIO sequence method to be applicable.

An FSM must satisfy certain constraints before it is used for constructing UIO sequences. First there is the "connected assumption" which implies that every state in the FSM is reachable from its initial state. The second

assumption is that the machine can be applied a *reset* input in any state that brings it to its start state. As has been explained earlier, a `null` output is generated upon the application of a reset input.

An FSM is assumed to remain in its current state and generate a null output when a transition on an input is not specified. Such an input is considered as non-core.

The third assumption is known as the *completeness* assumption. According to this assumption, an FSM remains in its current state upon the receipt of any input for which the state transition function δ is not specified. Such an input is known as a *non-core* input. In this situation, the machine generates a `null` output. The completeness assumption implies that each state contains a self-loop that generates a `null` output upon the receipt of an unspecified input. The fourth assumption is that the FSM must be minimal. A machine that depicts only the core behavior of an FSM is referred to as its core-FSM. The following example illustrates the core behavior.

Example 5.21 Consider the FSM in Figure 5.19(a) that is similar to the one in Figure 5.13. Note that states q_1, q_4, and q_5 have no transitions corresponding to inputs a, b, and b, respectively. Thus, this machine does not satisfy the completeness assumption.

As shown in Figure 5.19(b), additional edges are added to this machine in states q_1, q_4, and q_5. These edges represent transitions that generate `null` output. Such transitions are also known as erroneous transitions. Figure 5.19(a) shows the core behavior corresponding to the machine in Figure 5.19(b).

While determining the UIO sequences for a machine, only the core behavior is considered. The UIO sequences for all states of the machine shown in Figure 5.19(a) are given below.

State (s)	UIO (s)
q_1	b/1· a/1/· b/1· b/1
q_2	a/0· b/1
q_3	b/1· b/1
q_4	a/1· b/1· b/1
q_5	a/1· a/0· b/1

Note that the core behavior exhibited in Figure 5.19 is different from that in the original design in Figure 5.13. Self-loops that generate a `null` output, are generally not shown in a state diagram that depicts the core behavior. The impact of removing self-loops, that generate a non-null output, on the fault detection capability of

Figure 5.19 (a) An incomplete FSM. (b) Addition of `null` transitions to satisfy the completeness assumption.

The core-behavior of an FSM does not include self loops that generate null outputs.

the UIO method will be discussed in Section 5.8.9. In Figure 5.19, the set of core edges is $\{(q_1, q_4), (q_2, q_1), (q_2, q_5), (q_3, q_1), (q_3, q_5), (q_4, q_3), (q_5, q_2), (q_1, q_1), (q_2, q_2), (q_3, q_3), (q_4, q_4), (q_5, q_5)\}$.

During a test, one wishes to determine if the IUT behaves in accordance with its specification. An IUT is said to conform *strongly* to its specification if it generates the same output as its specification for all inputs. An IUT is said to conform *weakly* to its specification if it generates the same output as that generated by the corresponding core-FSM.

Example 5.22 Consider an IUT that is to be tested against its specification M as in Figure 5.13. Suppose that the IUT is started in state q_1 and the one symbol sequence a is input to it and the IUT outputs a `null` sequence, i.e. the IUT does not generate any output. In this case, the IUT does not conform strongly to M. However, on input a, this is exactly the behavior of the core-FSM. Thus, if the IUT behaves exactly as the core-FSM shown in Figure 5.19(a), then it conforms weakly to M.

5.8.4 Generation of UIO sequences

The UIO sequences are essential to the UIO-sequence method for generating tests from a given FSM. In this section, we present an algorithm for generating the UIO sequences. As mentioned earlier, a UIO sequence might not exist for one or more states in an FSM. In such a situation, a signature is used instead. An algorithm for generating UIO sequences for all states of a given FSM follows.

When a UIO sequence does not exist for some state s, another string known as its signature, is determined.

```
Procedure for generating UIO sequences
```

Input: (a) An FSM $M = (X, Y, Q, q_0, \delta, O)$ where $|Q| = n$.
(b) State $s \in Q$.

Output: $UIO(s)$ contains a unique input/output sequence for state s; $UIO(s)$ is empty if no such sequence exists.

Procedure: `gen-UIO(s)`

/* $Set(l)$ denotes the set of all edges with label l.
$label(e)$ denotes the label of edge e.
A label is of the kind a/b where $a \in X$ and $b \in Y$.
$head(e)$ and $tail(e)$ denote, respectively, the head and tail states of edge e. */

Step 1 For each distinct edge label el in the FSM, compute $Set(el)$.

Step 2 Compute the set *Oedges(s)* of all outgoing edges for state s. Let $NE = |Oedges|$.

Step 3 For $1 \leq i \leq NE$ and each edge $e_i \in Oedges$, compute $Oled[i]$, $Opattern[i]$, and $Oend[i]$ as follows:

3.1 $Oled[i] = Set(label(e_i)) - \{e_i\}$.
3.2 $Opattern[i] = label(e_i)$.
3.3 $Oend[i] = tail(e_i)$.

Step 4 Apply algorithm *gen-1-uio(s)* to determine if $UIO(s)$ consists of only one label.

Step 5 If simple UIO sequence found then return $UIO(s)$ and terminate this algorithm, otherwise proceed to the next step.

Step 6 Apply the *gen-long-uio(s)* procedure to generate longer $UIO(s)$.

Step 7 If longer UIO sequence found then return $UIO(s)$ and terminate this algorithm, else return with an empty $UIO(s)$

```
End of procedure gen-uio.
```

```
Procedure gen-1-uio
```

Input: State $s \in Q$.

Output: $UIO(s)$ of length 1 if it exists, an empty string otherwise.

Procedure: `gen-1-UIO(State s):`

Step 1 If $Oled[i] = \emptyset$ for $1 \leq i \leq NE$ then return $UIO(s) = label(e_i)$ otherwise return $UIO(s) =$ “”.

```
End of procedure gen-1-uio
```

`Procedure for generating longer UIO sequences`

Input: (a) *Oedges*, *Opattern*, *Oend*, and *Oled* as computed in *gen-uio*.

(b) State $s \in Q$.

Output: *UIO*(s) contains a unique input/output sequence for state s and is empty if no such sequence exists.

Procedure: `gen-long-uio (State s)`

Step 1 Let L denote the length of the *UIO* sequence being examined. Set $L = 1$. Let *Oedges* denote the set of outgoing edges from some state under consideration.

Step 2 Repeat steps below while $L < 2n^2$ or the procedure is terminated prematurely.

2.1 Set $L = L + 1$ and $k = 0$. Counter k denotes the number of distinct patterns examined as candidates for *UIO*(s). The following steps attempt to find a *UIO*(s) of size L.

2.2 Repeat steps below for $i = 1$ to *NE*. Index i is used to select an element of *Oend* to be examined next. Note that *Oend*[i] is the tail state of the pattern in *Opattern*[i].

2.2.1 Let *Tedges*(t) denote the set of incoming edges to state t. Compute *Tedges*(*Oend*[i]).

2.2.2 For each edge $te \in$ *Tedges* execute *gen-L-UIO(te)* until either all edges in *Tedges* have been considered or a *UIO*(s) is found.

2.3 Prepare for the next iteration. Set $k = NE$. For $1 \leq j \leq k$, set *Opattern*[j] = *Pattern*[j], *Oend*[j] = *End*[j], and *Oled*[j] = *Led*[j]. If the loop termination condition is not satisfied, then go back to Step 2.1 and search for UIO of the next higher length, else return to *gen-uio* indicating a failure to find any UIO for state s.

`End of procedure gen-long-uio`

`Procedure for generating UIO sequences of length L>1.`

Input: Edge *te* from procedure *gen-long-UIO* .

Output: *UIO*(s) contains a unique input/output sequence of length L for state s and is empty if no such sequence exists.

Procedure: `gen-L-uio ( edge te)`

Step 1 $k = k + 1$, *Pattern*[k] = *Opattarn*[k] · *label*(*te*). This could be a possible UIO.

Step 2 *End*[k] = *tail*(*te*) and *Led*[k] = ∅.

Step 3 For each pair $oe \in Oled[i]$, where $h = head(oe)$ and $t = tail(oe)$, repeat the next step.

3.1 Execute the next step for each edge $o \in Oedges(t)$.

3.1.1 If $label(o) = label(te)$ then
$Led[k] = Led[k] \bigcup \{(head(oe), tail(o))\}$.

Step 4 If $Led[k] = \emptyset$, then UIO of length L has been found. Set $UIO(s) = Pattern[k]$ and terminate this and all procedures up until the main procedure for finding UIO for a state. If $Led[k] \neq \emptyset$, then no UIO of length L has been found corresponding to edge te. Return to the caller to make further attempts.

```
End of Procedure gen-L-uio
```

5.8.5 Explanation of *gen-uio*

The gen-uio algorithm uses the gen-1-uio algorithm to generate a UIO of length 1. If such a UIO does not exist then it uses the gen-long-uio algorithm in an attempt to generate a longer UIO sequence. When a longer UIO sequence is not found, the algorithm terminates with an empty sequence.

The idea underlying the generation of a UIO sequence for state s can be explained as follows. Let M be the FSM under consideration and s a state in M for which a UIO is to be found. Let $E(s) = e_1 e_2 \ldots e_k$ be a sequence of edges in M such that $s = head(e_1), tail(e_i) = head(e_{i+1})$, and $1 \leq i < k$.

For $E(s)$, we define a string of edge labels as $label(E(s)) = l_1 \cdot l_2 \cdot \ldots l_{k-1} \cdot l_k$, where $l_i = label(e_i), 1 \leq i \leq k$, is the label of edge e_i. For a given integer $l > 0$, we find if there is a sequence $E(s)$ of edges starting from s such that $label(E(s)) \neq label(E(t)), s \neq t$ for all states t in M. If there is such an $E(s)$, then $label(E(s))$ is a UIO for state s. The uniqueness check is performed on $E(s)$ starting with $l = 1$ and continuing for $l = 2, 3, \ldots$ until a UIO is found or $l = 2n^2$, where n is the number of states in M. Note that there can be multiple UIOs for a given state though the algorithm generates only one UIO, if it exists.

The *gen-uio* algorithm is explained with reference to Figure 5.20. Algorithm *gen-uio* for state s begins with the computation of some sets used in the subsequent steps. First, the algorithm computes $Set(el)$ for all distinct labels el in the FSM. $Set(el)$ is the set of all edges that have el as their label.

Next, the algorithm computes the set $Oedges(s)$ which is the set of outgoing edges from state s. For each edge e in $Oedges(s)$, the set $Oled[e]$ is computed. $Oled[e]$ contains all edges in $Set(label(e))$ except for the edge e. The tail state of reach edge in $Oedges$ is saved in $Oend[e]$. The use of $Oled$ will become clear later in Example 5.27.

Figure 5.20 Flow of control across various procedures in *gen-uio*.

Example 5.23 To find $UIO(q_1)$ for machine M_1 in Figure 5.18, *Set*, *Oedges* and *Oled* are computed as follows.

Distinct labels in $M_1 = \{a/0, b/1, c/0, c/1\}$
$Set(a/0) = \{(1,2),(2,3),(3,4)_{a/0}\}$
$Set(b/1) = \{(3,4),(4,5)\}$
$Set(c/0) = \{(5,6)\}$
$Set(c/1) = \{(2,6),(6,1)\}$

$Oedges(q) = \{(1,2)\}$, $NE = 1$
$Oled[1] = \{(2,3),(3,4)_{a/0}\}$
$Oend[1] = q_2$
$Opattern[1] = a/0$

Next, *gen-1-uio* is called in an attempt to generate a UIO of length 1. For each edge $e_i \in Oedges$, *gen-1-uio* initializes $Opattern[i]$ to $label(e_i)$ and $Oend[i]$ to $tail(e_i)$. *Opattern* and *Oend* are used subsequently by *gen-uio* in case a UIO of length 1 is not found. *gen-1-uio* now checks if the label of any of the outgoing edges from s is unique. This is done by checking if $Oled[i]$ is empty for any $e_i \in Oedges(s)$. Upon return from *gen-1-uio*, *gen-uio* terminates if a UIO of length 1 was found, otherwise it invokes *gen-long-uio* in an attempt to find a UIO of length greater than 1. *gen-longer-uio* attempts to find UIO of length $L > 1$ until it finds one, or $L = 2n^2$.

Example 5.24 As an example of a UIO of length 1, consider state q_5 in Machine M_1 in Figure 5.18. The set of outgoing edges for state q_5 is $\{(5, 6)\}$. The label of $\{(5, 6)\}$ is $c/0$. Thus, from *gen-uio* and *gen-1-uio*, we get $Oedges(q_5) = \{(5,6)\}$, $Oled[1] = \emptyset$, $Oen[1] = q_6$, and $Opattern[1] = label(e_1) = c/0$. As $Oled[1] = \emptyset$, state q_5 has a UIO of length 1 which is $Opattern[1]$, i.e. $c/0$.

Example 5.25 As an example of a state for which a UIO of length 1 does not exist, consider state q_3 in Machine M_1 shown in Figure 5.18. For this state, we get the following sets:

$Oedges(q_3) = \{(3,4)_{a/0}, (3,4)_{b/1}\}$, $NE = 2$
$Oled[1] = \{(1,2),(2,3)\}$, $Oled[2] = \{(4,5)\}$

$Opattern[1] = a/0$, $Opattern[2] = b/1$
$Oend[1] = q_4$, $Oend[2] = q_4$
$Opattern[1] = a/0$, $Opattern[2] = b/1$

As no element of *Oled* is empty, *gen-1-uio* concludes that state q_3 has no UIO of length 1 and returns to *gen-uio*.

In the event *gen-1-uio* fails, procedure *gen-long-uio* is invoked. The task of *gen-long-uio* is to check if a $UIO(s)$ of length two or more exists. To do so, it collaborates with its descendent *gen-L-uio*. An incremental approach is used where UIOs of increasing length, starting at two, are examined until either a UIO is found or one is unlikely to be found, i.e. a maximum length pattern has been examined.

To check if there is UIO of length L, *gen-long-uio* computes the set $Tedges(t)$ for each edge $e \in Oedges(s)$, where $t = tail(e)$. For $L = 2$, $Tedges(t)$ will be a set of edges outgoing from the tail state t of one of the edges of state s. However, in general, $Tedges(t)$ will be the set of edges going out of a tail state of some edge outgoing from state s', where s' is a successor of s. After initializing *Tedges*, *gen-L-uio* is invoked iteratively for each element of $Tedges(t)$. The task of *gen-L-uio* is to find whether or not there exists a UIO of length L.

The gen-long-uio procedure is invoked when there is no UIO sequence of length 1.

Example 5.26 There is no $UIO(q_1)$ of length one. Hence *gen-long-uio* is invoked. It begins by attempting to find a UIO of length two, i.e. $L = 2$. From *gen-uio*, we have $Oedges(q_1) = \{(1,2)\}$. For the lone edge in *Oedges*, we get $t = tail((1,2)) = q_2$. Hence $Tedges(q_2)$, which is the set of edges out of state q_2, is computed as follows:

$Tedges(q_2) = \{(2,3),(2,6)\}$

gen-L-uio is now invoked first with edge (2, 3) as input to determine if there is a UIO of length two. If this attempt fails, then *gen-L-uio* is invoked with edge (2, 6) as input.

To determine if there exists a UIO of length L corresponding to an edge in *Tedges*, *gen-L-uio* initializes *Pattern*[k] by catenating the label of edge *te* to *Opattern*[k]. Recall that *Opattern*[k] is of length $(L-1)$ and has been rejected as a possible $UIO(s)$. *Pattern*[k] is of length L and is a candidate for $UIO(s)$. Next, the tail of *te* is saved in *End*[k]. Here, k serves as a running counter of the number of patterns of length L examined.

Next, for each element of *Oled*[i], *gen-L-uio* attempts to determine if indeed *Pattern*[k] is a $UIO(s)$. To understand this procedure, suppose that *oe* is an edge in *Oled*[i]. Recall that edge *oe* has the same label as the edge e_i and that e_i is not in *Oled*[i]. Let t be the tail state of *oe* and $Oedges(t)$ the set of all outgoing edges from state t. Then, *Led*[k] is the set of all pairs $(head(oe), tail(oe))$ such that $label(te) = label(oe)$ for all $o \in Oedges(t)$.

Note that an element of *Led*[k] is not an edge in the FSM. If *Led*[k] is empty after having completed the nested loops in Step 3, then *Pattern*[k] is indeed a $UIO(s)$. In this case, the *gen-uio* procedure is terminated and *Pattern*[*te*] is the desired UIO. If *Led*[k] is not empty, then *gen-L-uio* returns to the caller for further attempts at finding $UIO(s)$.

Example 5.27 Continuing Example 5.26, suppose that *gen-L-uio* is invoked with $i = 1$ and $te = (2,3)$. The goal of *gen-L-uio* is to determine if there is a path of length L starting from state $head(te)$ that has its label same as $Opattern[i] \cdot label(te)$.

Pattern[i] is set to $a/0 \cdot a/0$ because $Opattern[i] = a/0$ and $label((2,3)) = a/0$. *End*[i] is set to $tail(2,3)$ which is q_3. The outer loop in Step 3 examines each element *oe* of *Oled*[1]. Let $oe = (2,3)$ for which we get $h = q_2$ and $t = q_3$. The set $Oedges(q_3) = \{(3,4)_{a/0}, (3,4)_{b/1})\}$. Step 3.1 now iterates over the two elements of $OutEedges(q_3)$ and updates *Led*[i]. At the end of this loop, we have $Led[i] = \{(2,4)\}$ because $label((3,4)_{a/0}) = label((2,3))$.

Continuing with the iteration over *Oled*[1], *oe* is set to $(3,4)_{a/0}$ for which $h = q_2$ and $t = q_4$. The set $Oedges(q_4) = \{(4,5)\}$. Iterating over $Oedges(q_4)$ does not alter *Led*[i] because $label((4,5)) \neq label((3,4)_{a/0})$.

The outer loop in Step 3 is now terminated. In the next step, i.e. Step 4, *Pattern*[*i*] is rejected and *gen-L-uio* returns to *gen-long-uio*.

Upon return from *gen-L-uio*, *gen-long-uio* once again invokes it with $i = 1$ and $te = (2, 6)$. *Led*[2] is determined during this call. To do so, *Pattern*[2] is initialized to $a/0 \cdot c/1$ because $Opattern[i] = a/0$ and $label(2, 6) = c/1$, and *End*[2] to q_6.

Once again, the procedure iterates over elements *oe* of *Oled*. For $oe = (2, 3)$, we have as before, $Oedges(q_3) = \{(3, 4)_{a/0}, (3, 4)_{b/1}\}$. Step 3.1 now iterates over the two elements of $Oedges(q_3)$. In Step 1, none of the checks is successful as $label((2, 6))$ is not equal to $label((3, 4)_{a/0})$ or to $label((4, 5))$. The outer loop in Step 3 is now terminated. In the next step, i.e. Step 4, *Pattern*[2] is accepted as $UIO(q_1)$.

It important to note that *Led*[*i*] does not contain edges. Instead, it contains one or more pairs of states, (s_1, s_2) such that a path in the FSM from state s_1 to state s_2 has the same label as *Pattern*[*i*]. Thus, at the end of the loop in Step 3 of *gen-L-uio*, an empty *Led*[*i*] implies that there is no path of length L from $head(te)$ to $tail(te)$ with a label same as *Pattern*[*i*].

A call to *gen-long-uio* either terminates the *gen-uio* algorithm abruptly indicating that $UIO(s)$ has been found, or returns normally indicating that $UIO(s)$ is not found, and any remaining iterations should now be carried out. Upon a normal return from *gen-L-uio*, the execution of *gen-long-uio* resumes at Step 2.3. In this step, the existing *Pattern*, *Led*, and *End* data is transferred to *Opattern*, *Oled*, and *Oend*, respectively. This is done in preparation for the next iteration to determine if there exists a UIO of length $(L + 1)$. In case a higher length sequence remains to be examined, the execution resumes from Step 2.2, else the *gen-long-uio* terminates without having determined a $UIO(s)$.

Example 5.28 Consider M_2 in Figure 5.18. We invoke *gen-uio*(q_1) to find a UIO for state q_1. As required in Steps 1 and 2, we compute *Set* for each distinct label and the set of outgoing edges $Oedges(q_1)$.

Distinct labels in $M_2 = \{a/0, a/1, b/0, b/1\}$

$Set(a/0) = \{(1, 2), (3, 2)\}$
$Set(a/1) = \{(2, 1)\}$
$Set(b/0) = \{(1, 3), (2, 3)\}$
$Set(b/1) = \{(3, 1)\}$

$Oedges(q_1) = \{(1,2),(1,3)\}$
$NE = 2$

Next, as directed in Step 3, we compute *Oled* and *Oend* for each edge in $Oedges(q_1)$. The edges in *Oedges* are numbered sequentially so that edge (1, 2) is numbered 1 and edge (1, 3) is numbered 2.

$Oled[1] = \{(3,2)\}$
$Oled[2] = \{(2,3)\}$

$Oend[1] = q_2$
$Oend[2] = q_3$

$Opattern[1] = a/0$
$Opattern[2] = b/0$

gen-1-uio(q_1) is now invoked. As *Oled*[1] and *Oled*[2] are non-empty, *gen-1-uio* fails to find a UIO of length one and returns. We now move to Step 6 where procedure *gen-long-uio* is invoked. It begins by attempting to check if a UIO of length two exists. Towards this goal, the set *Tedges* is computed for each edge in *Oedges* at Step 2.2.1 of *gen-long-uio*. First, for $i = 1$, we have $Oled[1] = (3,2)$ whose tail is state q_2. Thus, we obtain $Tedges(q_2) = \{((2,3),(2,1)\}$.

The loop for iterating over the elements of *Tedges* begins at Step 2.2.2. Let $te = (2,3)$. *gen-L-uio* is invoked with *te* as input and the value of index i as 1. Inside *gen-L-uio*, k is incremented to 1 indicating that the first pattern of length two is to be examined. *Pattern*[1] is set to $a/0 \cdot b/0$, *End*[1] to 3, and *Led*[1] initialized to the empty set.

The loop for iterating over the two elements of *Oled* begins at Step 3. Let $oe = (3,2)$ for which $h = 3$, $t = 2$, and $Oedges(2) = \{(2,1),(2,3)\}$. We now iterate over the elements of *Oedges* as indicated in Step 3.1. Let $o = (2,1)$. As labels of (2, 1) and (2, 3) do not match, *Led*[1] remains unchanged. Next, $o = (2,3)$. This time we compare labels of o and *te*, which are the same edges. Hence, the two labels are the same. As directed in Step 3.1.1, we set $Led[1] = (head(oe), tail(o)) = (3,3)$. This terminates the iteration over *Oedges*. This also terminates the iteration over *Oled* as it contains only one element.

In Step 4, we find that *Led*[1] is not empty and therefore reject *Pattern*[1] as a UIO for state q_1. Notice that *Led*[1] contains a pair (3, 3) which implies that there is a path from state q_3 to q_3 with the label identical to that in *Pattern*[1]. A quick look at the state

diagram of M_2 in Figure 5.18 reveals that indeed the path $q_3 \rightarrow q_2 \rightarrow q_3$ has the label $a/0 \cdot b/0$ which is the same as in *Pattern*[1].

Control now returns to Step 2.2.2 in *gen-long-uio*. The next iteration over $Tedges(q_2)$ is initiated with $te = (2, 1)$. *gen-L-uio* is invoked once again but this time with index $i = 1$ and $te = (2, 1)$. The next pattern of length two to be examined is $Pattern[2] = a/0 \cdot a/1$. We now have $End[2] = 1$ and $Led[2] = \emptyset$. Iteration over elements of *Oled*[1] begins. Once again let $oe = (3, 2)$ for which $h = 3$, $t = 2$, and $Oedges(2) = \{(2, 1), (2, 3)\}$. Iterating over the two elements of *Oeges* we find that the label of *te* matches that of edge (2, 1) and not of edge (2, 3). Hence, we get $Led[2] = (head(oe), tail(2, 1)) = (3, 1)$ implying that *Pattern*[2] is the same as the label of the path from state q_3 to state q_1. The loop over *Oled* also terminates and control returns to *gen-long-uio*.

In *gen-long-uio*, the iteration over *Tedges* now terminates as we have examined both edges in *Tedges*. This also completes the iteration for $i = 1$ which corresponds to the first outgoing edge of state q_1. We now repeat Step 2.2.2 for $i = 2$. In this iteration, the second outgoing edge of state q_1, i.e. edge (1, 3), is considered. Without going into the fine details of each step, we leave it to you to verify that at the end of the iteration for $i = 2$, we get the following.

$Pattern[3] = b/0 \cdot a/0$
$Pattern[4] = b/0 \cdot b/1$
$End[3] = 2$
$End[4] = 1$
$Led[3] = (2, 2)$
$Led[4] = (2, 1)$

This completes the iteration set up in Step 2.2. We now move to Step 3. This step leads to the following values of *Opattern*, *Oled*, and *Oend*.

$Opattern[1] = Pattern[1] = a/0 \cdot b/0$
$Opattern[2] = Pattern[2] = a/0 \cdot a/1$
$Opattern[3] = Pattern[3] = b/0 \cdot a/0$
$Opattern[4] = Pattern[4] = b/0 \cdot b/1$

$Oend[1] = End[1] = 3$
$Oend[2] = End[2] = 1$
$Oend[3] = End[3] = 2$
$Oend[4] = End[4] = 1$

$Oled[1] = Led[1] = (3, 3)$
$Oled[2] = Led[2] = (3, 1)$

$Oled[3] = Led[3] = (2, 2)$
$Oled[4] = Led[4] = (2, 1)$

The while-loop continues with the new values of *Opattern*, *Oend*, and *Oled*. During this iteration $L = 3$ and hence patterns of length three will be examined as candidates for $UIO(q_1)$. The patterns to be examined next will have at least one pattern in *Opattern* as its prefix. For example, one of the patterns examined for $i = 1$ is $a/0 \cdot b/0 \cdot a/0$.

The *gen-L-uio* procedure is invoked first with $e = (1, 2)$ and $te = (2, 3)$ in Step 2.2.2. It begins with $Pattern[1] = a/0 \cdot b/0$ and $End[1] = 3$.

Example 5.29 This example illustrates the workings of procedure *gen-uio(s)* by tracing through the algorithm for Machine M_1 in Figure 5.18. The complete set of UIO sequences for this machine is given in Example 5.19. In this example, we sequence through *gen-uio(s)* for $s = q_1$. The trace follows.

---

gen-uio *Input*: State q_1

Step 1 Find the set of edges for each distinct label in M_1. There are four distinct labels in M_1, namely $a/0, b/1, c/0$, and $c/1$. The set of edges for each of these four labels is given below.

$Set(a/0) = \{(1, 2), (2, 3), (3, 4)\}$
$Set(b/1) = \{(3, 4), (4, 5)\}$
$Set(c/0) = \{(5, 6)\}$
$Set(c/1) = \{(2, 6), (6, 1)\}$

Step 2 It is easily seen from the state transition function of M_1 that the set of outgoing edges from state q_1 is $\{(1,2)\}$. Thus, we obtain $Oedges(q_1) = \{(1, 2)\}$ and $NE = 1$.

Step 3 For each edge in *Oedges*, we compute *Oled*, *Opattern*, and *Oend*.

$Oled[1] = \{(1, 2), (2, 3), (3, 4)_{a/0}\} - \{(1, 2)\} = \{(2, 3), (3, 4)_{a/0}\}$.
$Opattern[1] = label((1, 2)) = a/0$, and
$Oend[1] = tail((1, 2)) = 2$.

Step 4 Procedure *gen-1-UIO* is invoked with state q_1 as input. In this step, an attempt is made to determine if there exists a UIO sequence of length 1 for state q_1.

---

gen-1-UIO *Input*: State q_1

Step 1 Now check if any element of *Oled* contains only one edge. From the computation done earlier, note that there is only one element in *Oled*, and that this element, *Oled*[1], contains two edges.

Hence, there is no $UIO(q_1)$ with only a single label. The *gen-1-UIO* procedure is now terminated and the control returns to *gen-UIO*.

---

gen-uio *Input*: State q_1

Step 5 Determine if a simple UIO sequence was found. As it has not been found, move to the next step.

Step 6 Procedure *gen-long-UIO* is invoked in an attempt to generate a longer $UIO(q_1)$.

---

gen-long-UIO *Input*: State q_1

Step 1 $L = 1$.

Step 2 Start a loop to determine if a $UIO(q_1)$ of length greater than 1 exists. This loop terminates when a UIO is found or when $L = 2n^2$, which for a machine with $n = 6$ states translates to $L = 72$. Currently, L is less than 72 and hence continue with the next step in this procedure.

Step 2.1 $L = 2$ and $k = 0$.

Step 2.2 Start another loop to iterate over the edges in *Oedges*. Set $i = 1$ and $e_i = (1, 2)$.

Step 2.2.1 $t = tail((1, 2)) = 2$. $Tedges(2) = \{(2, 3), (2, 6)\}$.

Step 2.2.2 Yet another loop begins at this point. Loop over the elements of *Tedges*. First set $te = (2, 3)$ and invoke *gen-L-UIO*.

---

gen-L-UIO *Input*: $te = (2, 3)$

Step 1 $k = 1$, $Pattern[1] = Opattern[1] \cdot label((2, 3)) = a/0 \cdot a/0$.

Step 2 $End[1] = 3$, $Led[1] = \emptyset$.

Step 3 Now iterate over elements of $Oled[1] = \{(2, 3), (3, 4)_{a/0}\}$. First select $oe = (2, 3)$ for which $h = 2$ and $t = 3$.

Step 3.1 Another loop is set up to iterate over elements of $Oedges(q_3) = \{(3, 4)_{a/0}, (3, 4)_{b/1}\}$. Select $o = (3, 4)_{a/0}$.

Step 3.1.1 As $label((3, 4)_{a/0}) = label((2, 3))$, set $Led[1] = \{(2, 4)\}$.
Next select $o = (3, 4)_{b/1}$ and execute Step 3.1.1.

Step 3.1.1 As $label((2, 3)) \neq label((3, 4)_{b/1})$, no change is made to $Led[1]$.
The iteration over *Oedges* terminates.
Next, continue from Step 3.1 with $oe = (3, 4)_{a/0}$ for which $h = 3$ and $t = 4$.

Step 3.1 Another loop is set up to iterate over elements of $Oedges(4) = \{(4, 5)\}$, select $o = (4, 5)$.

Step 3.1.1 As $label((4, 5)) \neq label((3, 4)_{a/0})$ no change is made to $Led[1]$.
The iterations over *Oedges* and $Oled[1]$ terminate.

Step 4 $Led[1]$ is not empty which implies that this attempt to find a $UIO(q_1)$ has failed. Note that in this attempt the algorithm checked if $a/0 \cdot a/0$ is a valid $UIO(q_1)$. Now return to the caller.

---

gen-long-UIO *Input*: State q_1

Step 2.2.2 Select another element of *Tedges* and invoke *gen-L-UIO (e, te)*. For this iteration $te = (2, 6)$.

---

gen-L-UIO	*Input* : $te = (2, 6)$
Step 1	$k = 2$, $Pattern[2] = Opattern[1] \cdot label((2, 6)) = a/0 \cdot c/1$.
Step 2	$End[2] = 6$, $Led[2] = \emptyset$.
Step 3	Iterate over elements of $Oled[1] = \{(2, 3), (3, 4)_{a/0}\}$. First select $oe = (2, 3)$ for which $h = 2$ and $t = 3$.
Step 3.1	Another loop is set up to iterate over elements of $Oedges(3) = \{(3, 4)_{a/0}, (3, 4)_{b/1}\}$, select $o = (3, 4)_{a/0}$.
Step 3.1.1	As $label((3, 4)_{a/0}) \neq label((2, 6))$, do not change $Led[2]$. Next select $o = (3, 4)_{b/1}$ and once again execute Step 3.1.1.
Step 3.1.1	As $label((2, 6)) \neq label((3, 4)_{b/1})$, do not change $Led[2]$. The iteration over *Oedges* terminates. Next continue from Step 3.1 with $oe = (3, 4)_{a/0}$ for which $h = 3$ and $t = 4$.
Step 3.1	Another loop is set up to iterate over elements of $Oedges(4) = \{(4, 5)\}$, select $o = (4, 5)$.
Step 3.1.1	As $label((4, 5) \neq label((2, 6))$ no change is made to $Led[2]$. The iterations over *Oedges* and $Oled[2]$ terminate.
Step 4	$Led[2]$ is empty. Hence $UIO(q_1) = Pattern[2] = a/0 \cdot c/1$. A *UIO* of length 2 for state q_1 is found and hence the algorithm terminates.

5.8.6 Distinguishing signatures

As mentioned earlier, the *gen-uio* procedure might return an empty $UIO(s)$ indicating that it failed to find a UIO for state s. In this case, we compute a signature that distinguishes s from other states one by one. We use $Sig(s)$ to denote a signature of state s. Before we show the computation of such a signature, we need some definitions. Let $W(q_i, q_j), i \neq j$ be a sequence of edge labels that distinguishes states q_i and q_j. Note that $W(q_i, q_j)$ is similar to the distinguishing sequence W for q_i and q_j except that we are now using edge labels of the kind a/b, where a is an input symbol and b an output symbol, instead of using only the input symbols.

A distinguishing sequence, or a signature, of state s is a sequence of input/output labels that are unique to s. In a sense this is a UIO sequence for s but computed using a different procedure than gen-uio.

Example 5.30 A quick inspection of M_2 in Figure 5.18 reveals the following distinguishing sequences.

$W(q_1, q_2) = a/0$
$W(q_1, q_3) = b/0$
$W(q_2, q_3) = b/0$

To check if indeed the above are correct distinguishing sequences, consider states q_1 and q_2. From the state transition function of M_2, we get $O(q_1, a) = 0 \neq O(q_2, a)$. Similarly, $O(q_1, b) = 0 \neq O(q_3, b)$,

$O(q_2, a) \neq O(q_3, a)$ and $O(q_2, b) \neq O(q_3, b)$. For a machine more complex than M_2, one can find $W(q_i, q_j)$ for all pairs of states q_i and q_j using the algorithm in Section 5.5 and using edge labels in the sequence instead of the input symbols.

A transfer sequence for a pair of states (r, s) is a sequence of edge labels that correspond to the shortest path from state s to state r.

We use $P_i(j)$ to denote a sequence of edge labels along the shortest path from state q_j to q_i. $P_i(j)$ is known as a *transfer sequence* for state q_j to move the machine to state q_i. When the inputs along the edge labels of $P_i(j)$ are applied to a machine in state q_j, the machine moves to state q_i. For $i = j$, the null sequence is the transfer sequence. As we see later, $P_i(j)$ is used to derive a signature when the *gen-uio* algorithm fails.

Example 5.31 For M_2, we have the following transfer sequences.

$P_1(q_2) = a/1$
$P_1(q_3) = b/1$
$P_2(q_1) = a/0$
$P_2(q_3) = a/0$
$P_3(q_1) = b/0$
$P_3(q_2) = b/0$

For M_1 in Figure 5.18, we get the following subset of transfer sequences (you may derive the others by inspecting the transition function of M_1).

$P_1(q_5) = c/0 \cdot c/1$
$P_5(q_2) = a/0 \cdot a/0 \cdot b/1$ or $P_5(2) = a/0 \cdot b/1 \cdot b/1$
$P_6(q_1) = a/0 \cdot c/1$

A transfer sequence $P_i(j)$ can be found by finding the shortest path from state q_j to q_i and catenating in order the labels of the edges along this path.

To understand how the signature is computed, suppose that *gen-uio* fails to find a UIO for some state $q_i \in Q$ where Q is the set of n states in machine M under consideration. The signature for s consists of two parts. The first part of the sequence is $W(q_i, q_1)$ which distinguishes q_i from q_1.

Now suppose that the application of sequence $W(q_i, q_1)$ to state q_i takes the machine to some state t_1. The second part of the signature for q_i consists of pairs $P_i(t_k) \cdot W(q_i, q_{k+1})$ for $1 \leq k < n$. Notice that the second part can be further split into two sequences.

The first sequence, $P_i(t_k)$, transfers the machine from t_k back to q_i. The second sequence, $W(q_i, q_{k+1})$, applies a sequence that distinguishes q_i from state q_{k+1}. Thus, in essence, the signature makes use of the sequences that distinguish q_i from all other states in the machine and the transfer

sequences that move the machine back to state q_i prior to applying another distinguishing sequence. Given that $q_1 \in Q$ is the starting state of M, a compact definition of the signature for state $q_i \in Q$ follows.

$$\begin{aligned} Sig(q_i) &= W(q_1, q_2) \cdot (P_1(t_2) \cdot W(q_1, q_3)) \cdot (P_1(t_3) \cdot W(q_1, q_4)) \ldots \cdot \\ &\quad (P_1(t_{n-1}) \cdot W(q_1, q_n)), \quad \text{for } i = 1. \\ &= W(q_i, q_1) \cdot (P_i(t_1) \cdot W(q_i, q_2)) \cdot (P_i(t_2) \cdot W(q_i, q_3)) \ldots \cdot \\ &\quad (P_i(t_{i-2}) \cdot W(q_i, q_{i-1})) \cdot (P_i(t_{i-1}) \cdot W(q_i, q_{i+1})) \ldots \cdot \\ &\quad (P_i(t_{n-1}) \cdot W(q_i, q_n)), \quad \text{for } i \neq 1. \end{aligned}$$

Example 5.32 We have seen earlier that *gen-long-uio* fails to generate a UIO sequence for state q_1 in M_2 shown in Figure 5.18. Let us therefore apply the method described above for the construction of a signature for state q_1. The desired signature can be found by substituting the appropriate values in the following formula.

$$Sig(q_1) = W(q_1, q_2) \cdot (P_1(t_1) \cdot W(q_1, q_3))$$

From Example 5.30, we have the following distinguishing sequences for state q_1.

$$W(q_1, q_2) = a/0$$
$$W(q_1, q_3) = b/0$$

The application of $W(q_1, q_2)$ to state q_1 takes M_2 to state q_2. Hence we need the transfer sequence $P_1(q_2)$ to bring M_2 back to state q_1. From Example 5.31 we get $P_1(q_2) = a/1$. Substituting these values in the formula for $Sig(q_1)$, we obtain the desired signature.

$$Sig(q_1) = a/0 \cdot a/1 \cdot b/0$$

Later, while deriving test cases from UIO of different states, we will use signatures for states that do not possess a UIO.

5.8.7 Test generation

Let $M = (X, Y, Q, q_1, \delta, O)$ be an FSM from which we need to generate tests to test an implementation for conformance. Let E denote the set of core edges in M. m is the total number of core edges in M. Recall that edges corresponding to a reset input in each state are included in E. The following procedure is used to construct a total of m tests, each corresponding to the *tour* of an edge.

An non-core edge in an FSM is any edge that is not a self loop that generates a null output.

1. Find the UIO for each state in M.
2. Find the shortest path from the initial state to each of the remaining states. As before, the shortest path from the initial state q_1 to any other state $q_i \in Q$ is denoted by $P_i(q_1)$.
3. Construct an *edge tour* for each edge in M. Let $TE(e)$ denote a subsequence that generates a tour for edge e. $TE(e)$ is constructed as follows

$$TE(e) = P_{head(e)}(1) \cdot label(e) \cdot UIO(tail(e)).$$

4. This step is optional. It is used to combine the M test subsequences generated in the previous step into one test sequence that generates the tour of all edges. This sequence is denoted by TA. It is sometimes referred to as β-sequence. TA is obtained by catenating pairs of reset input and edge tour subsequences as follows

   $TA = X_{e \in E}((Re/\texttt{null}) \cdot TE(e)).$

TA is useful when the IUT can be brought automatically to its start state by applying an Re. In this case, the application of TA will likely shorten the time to test the IUT. While testing an IUT, the application of Re might be possible automatically through a script that sends a "kill process" signal to terminate the IUT and, upon the receipt of an acknowledgment that the process has terminated, may restart the IUT for the next test.

The next example illustrates the test generation procedure using the UIO sequences generated from M_1 shown in Figure 5.18.

Example 5.33 The UIO sequences for each of the six states in M_1 are reproduced below for convenience. Also included in the rightmost column are the shortest paths from state q_1 to the state corresponding to the state in the leftmost column.

State(s)	UIO (s)	$P_i(q_1)$
q_1	a/0· c/1	`null`
q_2	c/1· c/1	a/0
q_3	b/1· b/1	a/0· a/0
q_4	b/1· c/0	a/0· a/0· a/0
q_5	c/0	a/0· a/0· a/0· b/1
q_6	c/1· a/0	a/0· c/1

We consider only the core edges while developing tests for each edge. For example, the self-loop in state q_5 corresponding to inputs a and b, not shown in Figure 5.18, is ignored as it is not part of the core behavior of M_1. Also note that edge (q_6, q_1) will be considered twice, one corresponding to the label $c/0$ and the other corresponding to label *Re*/`null`. Using the formula given above, we obtain the following tests for each of the 14 core edges.

Test count	Edge (e)	*TE*(*e*)
1	q_1, q_2	a/0· c/1· c/1
2	q_1, q_1	*Re*/`null`/· *Re*/`null`/· a/0· c/1
3	q_2, q_3	a/0· a/0· b/1· b/1
4	q_2, q_6	a/0· c/1· c/1· a/0
5	q_2, q_1	a/0· *Re*/`null`· a/0· c/1
6	q_3, q_4	a/0· a/0· a/0· b/1· c/0
7	q_3, q_4	a/0· a/0· b/1· b/1· c/0
8	q_3, q_1	a/0· a/0· *Re*/`null`· a/0· c/1
9	q_4, q_5	a/0· a/0· a/0· c/0
10	q_4, q_1	a/0· a/0· a/0· *Re*/`null`· a/0· c/1
11	q_5, q_6	a/0· a/0· b/1· c/0· c/1
12	q_5, q_1	a/0· a/0· a/0· b/1· *Re*/`null`· a/0· c/1
13	q_6, q_1	a/0· c/1· c/1· a/0· c/1
14	q_6, q_1	a/0· c/1· *Re*/`null`· a/0· c/1

The 14 tests derived above can be combined into a β-sequence and applied to the IUT. This sequence will exercise only the core edges. Thus, for example, the self-loops in state q_5, corresponding to inputs a and b will not be exercised by the tests given above. It is also important to note that each *TE*(*e*) subsequence is applied with the IUT in its start state implying that a *Re* input is applied to the IUT prior to exercising it with the input portion of *TE*(*e*).

Tests generated using the UIO method can be reduced using a simple optimization procedure.

5.8.8 Test optimization

The set of test subsequences *TE*(*e*) can often be reduced by doing a simple optimization. For example, if $TE(e_1)$ is a subsequence of test $TE(e_2)$, then $TE(e_1)$ is redundant. This is because the edges toured by $TE(e_1)$ are also

toured by $TE(e_2)$, and in the same order. Identification and elimination of subsequences that are fully contained in another subsequence generally leads to a reduction in the size of the test suite. In addition, if two tests are identical, then one of them can be removed from further consideration. (See Exercise 5.21.)

Example 5.34 In an attempt to reduce the size of the test set derived in Example 5.33, we examine each test and check if it is contained in any of the remaining tests. We find that test 3 is fully contained in test 7, test 1 is contained in test 4, and test 4 is contained in test 13. Thus, the reduced set of tests consists of 11 tests: 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, and 14 given in Example 5.33.

The tests derived in Example 5.33 are useful for a weak conformance test. To perform a strong conformance test of the IUT against the specification, we need to derive tests for non-core edges as well. The method for deriving such tests is the same as that given earlier for deriving $TE(e)$ for the tour of edge e except that e now includes non-core edges.

Example 5.35 We continue Example 5.33 for M_1 and derive additional tests needed for strong conformance. To do so, we need to identify the non-core edges. There are 10 non-core edges corresponding to the six states. Figure 5.21 shows the state diagram of M_1 with both core and non-core edges shown. Tests that tour the non-core edges can be generated easily using the formula for TE given earlier. The 10 tests follow.

Test count	Edge (e)	$TE(e)$
1	$(q_1, q_1)_b$/`null`	b/`null`· a/0· c/1
2	$(q_1, q_1)_c$/`null`	c/`null`· a/0· c/1
3	$(q_2, q_2)_b$/`null`	a/0· b/`null`· c/1· c/1
4	$(q_3, q_3)_c$/`null`	a/0· a/0· c/`null`· b/1· b/1
5	$(q_4, q_4)_a$/`null`	a/0· a/0· a/0· a/`null`· b/1· c/0
6	$(q_4, q_4)_c$/`null`	a/0· a/0· a/0· c/`null`· b/1· c/0
7	$(q_5, q_5)_a$/`null`	a/0· a/0· a/0· b/1· a/`null`· c/0
8	$(q_5, q_5)_b$/`null`	a/0· a/0· a/0· b/1· b/`null`· c/0
9	$(q_6, q_6)_a$/`null`	a/0· c/1· a/`null`· c/1· a/0
10	$(q_6, q_6)_b$/`null`	a/0· c/1· b/`null`· c/1· a/0

Figure 5.21 Machine M_1 from Figure 5.18 with all non-core edges shown as dashed lines.

Note that a test for non-core edge e is similar to the tests for core edges in that the test first moves the machine to $head(e)$. It then traverses the edge itself. Finally, it applies $UIO(tail(e))$ starting at state $tail(e)$. Thus, in all, we have 21 tests to check for strong conformance of an IUT against M.

5.8.9 Fault detection

Tests generated using the UIO sequences are able to detect all operation and transfer errors. However, combination faults, such as an operation error and a transfer error might not be detectable. The next two examples illustrate the fault detection ability of the UIO method.

Tests generated using the UIO sequence method are able to detect all operation and transfer errors. However, errors that are a combination of operation and transfer errors might not be detected by such tests.

Example 5.36 Consider the transition diagram in Figure 5.22. Suppose that this diagram represents the state transitions in an IUT to be tested for strong conformance against machine M_1 in Figure 5.21. The IUT has two errors. State q_2 has a transfer error

Figure 5.22 State diagram of an IUT containing two faults. State q_2 has a transfer error and q_3 has an operation error. The behavior of this IUT is to be tested against that of the machine in Figure 5.21. Non-core edges are not shown.

due to which $\delta(q_2, c) = q_5$ instead of $\delta(q_2, c) = q_6$. State q_3 has an operation error due to which $O(q_3, b)$ is undefined.

To test the IUT against its specification as in Figure 5.21, one needs to apply the β sequence to the IUT and observe its behavior. The β sequence is obtained by combining all subsequences derived for weak and strong conformance in the earlier examples. However, as such a β sequence is too long to be presented here, we use an alternate method to show how the faults will be detected.

First, consider the transfer error in state q_2. Let us apply the input portion of test 4, which is $TE(q_2, q_6)$, to the IUT. We assume that the IUT is in its start state, i.e. q_1, prior to the application of the test. From Example 5.33, we obtain the input portion as *acca*. The expected behavior of the IUT for this input is determined by tracing the behavior of the machine in Figure 5.21. Such a trace gives us $O(q_1, acca) = 0110$. However, when the same input is applied to the IUT, we obtain $O(q_1, acca) = 010\texttt{null}$.

As the IUT behaves differently than its specification, $TE(q_2, q_6)$ has been able to discover the fault. Note that if test 4 has been excluded due to optimization, then test 13 can be used instead. In this case, the expected behavior is $O(q_1, accac) = 01101$, whereas the IUT generates $O(q_1, accac) = 010\texttt{null}1$ which also reveals a fault in the IUT.

Next, let us consider the operation error in state q_3 along the edge $(q_3, q_4)_{c/1}$. We use test 7. The input portion of this test is *aabbc*. For the specification, we have $O(q_1, aabbc) = 00110$. Assuming that this test is applied to the IUT in its start state, we get $O(q_1, aabbc) = 00\texttt{null}\texttt{null}1$ which is different from the expected output and thus test 7 has revealed the operation fault. (Also see Exercise 5.10.)

Example 5.37 Consider the specification shown in Figure 5.23(a). We want to check if tests generated using the UIO method will be able to reveal the transfer error in the IUT shown in Figure 5.23(b). UIO sequences and shortest paths from the start state to the remaining two states are given below.

State(s)	UIO (s)	$P_i(q_1)$
q_1	a/1	null
q_2	a/0· a/1	a/0
q_3	b/1· a/1	b/1
	(also a/0· a/0)	

Figure 5.23 State diagram of an FSM for which a test sequence derived using the UIO approach does not reveal the error. (a) Specification FSM. (b) State diagram of a faulty IUT.

Given below are tests for touring each of the nine core edges including three edges that bring each state to state q_1 upon reset. Notice that we have added the *Re*/`null` transition at the start of each test sequence to indicate explicitly that the machine starts in state q_1 prior to the test being applied.

Test count	Edge (e)	*TE*(*e*)
Edges (e) from each state to state q_1, label(e) = Re/`null`		
1	q_1, q_1	*Re*/`null`· *Re*/`null`· a/1
2	q_2, q_1	*Re*/`null`· a/1*Re*/`null`· a/1
3	q_3, q_1	*Re*/`null`· b/1· *Re*/`null`· a/1
Edges shown in Figure 5.23(a).		
4	q_1, q_2	*Re*/`null`· a/1· a/0· a/1
5	q_1, q_3	*Re*/`null`· b/1· b/1· a/1
6	q_2, q_1	*Re*/`null`· a/1· a/0· a/1
7	q_2, q_3	*Re*/`null`· a/1· b/1· b/1· a/1
8	q_3, q_1	*Re*/`null`· b/1· b/1· a/1
9	q_3, q_2	*Re*/`null`· b/1· a/0· a/0· a/1

To test the IUT let us apply the input portion *bba* of test 5 that tours the edge (q_1, q_3). The expected output is 111. The output generated by the IUT is also 111. Hence, the tour of edge (q_1, q_3) fails to reveal the transfer error in state q_1. However, test 9 that tours edge (q_3, q_2) does reveal the error because the IUT generates the output 1101, whereas the expected output is 1001, ignoring the `null` output.

Note that tests 4 and 6 are identical and so are tests 5 and 8. Thus, an optimal test sequence for this example is the set containing tests 1, 2, 3, 4, 5, 7, and 9.

5.9 Automata Theoretic Versus Control-Flow Based Techniques

The test generation techniques described in this chapter fall under the *automata theoretic* category. There exist other techniques that fall under the *control-flow based* category. Here, we compare the fault detection effectiveness of some techniques in the two categories.

Several empirical studies have aimed at assessing the fault detection effectiveness of test sequences generated from FSMs by various test generation methods. Here we compare the fault detection effectiveness of the W and Wp-methods with four control-flow based criteria to assess the adequacy of tests. Some of the control theoretic can be applied to assess the adequacy of tests derived from FSMs against the FSM itself. In this section, we define four such criteria and show that tests derived from the W- and Wp-methods are superior in terms of their fault detection effectiveness than the four control-flow based methods considered.

Tests generated using the W- and Wp-methods guarantee the detection of all missing transitions, incorrect transitions, extra or missing states, and errors in the output associated with a transition given that the underlying assumptions listed in Section 5.6.1 hold. We show through an example that tests generated using these methods are more effective in detecting faults than the tests that are found adequate with respect to *state cover*, *branch cover*, *switch cover*, and the *boundary interior* cover test adequacy criteria. First, a few definitions before we illustrate this fact.

The state cover, transition cover, and the switch cover are control theoretic criteria for assessing the adequacy of tests generated from FSMs. The W, Wp, and the UIO methods are automata theoretic.

State cover:

A test set T is considered adequate with respect to the state cover *criterion for an FSM M if the execution of M against each element of T causes each state in M to be visited at least once.*

Transition cover:

A test set T is considered adequate with respect to the branch*, or transition, cover criterion for an FSM M if the execution of M against each element of T causes each transition in M to be taken at least once.*

Switch cover:

A test set T is considered adequate with respect to the 1-switch cover *criterion for an FSM M if the execution of M against each element of T causes each pair of transitions* (tr_1, tr_2) *in M to be taken at least once, where for*

some input substring $ab \in X^*$, $tr_1 : q_i = \delta(q_j, a)$ *and* $tr_2 : q_k = \delta(q_i, b)$ *and* q_i, q_j, q_k *are states in M.*

Boundary-interior cover:

A test set T is considered adequate with respect to the boundary-interior cover *criterion for an FSM M if the execution of M against each element of T causes each loop body to be traversed zero times and at least once. Exiting the loop upon arrival covers the "boundary" condition and entering it and traversing the body at least once covers the "interior" condition.*

The next example illustrates weaknesses of the state cover, branch cover, switch cover, and the boundary interior cover test adequacy criteria.

Example 5.38 This example is due to T.S. Chow. Machine M_1 in Figure 5.24 represents the correct design while M_1' has an operation and a transfer error in state q_2. Note that M_1 has two states and four transitions.

Consider the input sequence $t = aabb$. t covers all states and transitions in M_1 and hence is adequate with respect to the state coverage and transition coverage criteria. We obtain $\delta_{M1}(q_1, t) = 0111$ and $\delta_{M1'}(q_1, t) = 0100$. Thus, the operation error is detected but not the transfer error.

Next, consider machine M_2 in Figure 5.25 that represents correct design while M_2' has a transfer error in state q_3. In order for a test set to be adequate with respect to the switch cover criterion, it must cause the following set of branch pairs to be exercised.

$$S = \{(tr_1, tr_2), (tr_1, tr_3), (tr_2, tr_2), (tr_2, tr_3), (tr_3, tr_4), (tr_3, tr_5), (tr_4, tr_4), (tr_4, tr_5), (tr_5, tr_6), (tr_5, tr_1), (tr_6, tr_4), (tr_6, tr_5)\}$$

Figure 5.24 Machine M_1' contains a transfer fault with respect to machine M_1 in state q_2.

Figure 5.25 Machine M_2' contains a transfer fault with respect to machine M_2 in state q_3.

The following table lists a set of test sequences adequate with respect to the switch cover criterion but do not reveal the transfer error in state q_3. The second column in the table shows the output generated and the rightmost column lists the switches covered.

Test sequence (t)	$O_{M_2}(q_1,t)$ $(=O_{M_2'}(q_1, t))$	**Switches covered**
abbaaab	0111001	$(tr_1, tr_2), (tr_2, tr_2), (tr_2, tr_3),$ $(tr_3, tr_4), (tr_4, tr_4), (tr_4, tr_5)$
aaba	0110	$(tr_1, tr_3), (tr_3, tr_5), (tr_5, tr_1)$
aabb	0110	$(tr_1, tr_3), (tr_3, tr_5), (tr_5, tr_6)$
baab	0001	$(tr_6, tr_4), (tr_4, tr_4), (tr_4, tr_5)$
bb	01	(tr_6, tr_5)

A simple inspection of machine M_2 in Figure 5.25 reveals that all states are 1-distinguishable, i.e. for each pair of states (q_i, q_j), there exists a string of length 1 that distinguishes q_i from q_j, $1 \leq (i,j) \leq 3$, $i \neq j$. Later we define an n-switch cover and show how to construct one that reveals all transfer and operation errors for an FSM that is n-distinguishable.

5.9.1 *n*-switch-cover

The switch-cover criterion can be generalized to an n-switch cover criterion as follows. An *n-switch* is a sequence of $(n+1)$ transitions. For example, for machine M_3 in Figure 5.26, transition sequence tr_1 is a 0-switch, tr_1, tr_2 is a 1-switch, tr_1, tr_2, tr_3 is a 2-switch, and transition sequence tr_6, tr_5, tr_1, tr_3 is a 3-switch. For a given integer $n > 0$, we can define an n-switch set for a transition tr as the set of all n-switches with tr as the prefix. For example, for each of the six transitions in Figure 5.26, we have the following 1-switch sets.

An *n*-switch is a sequence of $(n+1)$ transitions. In test set derived for an FSM, M is considered to be adequate with respect to the *n*-switch cover criterion if it covers all *n*-switches in M.

$tr_1 : \{(tr_1, tr_2), (tr_1, tr_3)\}$

$tr_2 : \{(tr_2, tr_2), (tr_2, tr_3)\}$

$tr_3 : \{(tr_3, tr_4), (tr_3, tr_5)\}$

$tr_4 : \{(tr_4, tr_4), (tr_4, tr_5)\}$

$tr_5 : \{(tr_5, tr_6), (tr_5, tr_1)\}$

$tr_6 : \{(tr_6, tr_4), (tr_6, tr_5)\}$

An n-switch set S for a transition tr in FSM M is considered covered by a set T of test sequences if exercising M against elements of T causes each transition sequence in S to be traversed. T is considered an *n-switch set cover*, if it covers all n-switch sets for FSM M. It can be shown that an n-switch set cover can detect all transfer, operation, extra, and missing state errors in a minimal FSM that is n-distinguishable (see Exercise 5.28). Given a minimal, 1-distinguishable FSM M, the next example demonstrates a procedure to obtain a 1-switch cover using the testing tree of M.

Figure 5.26 Machine M_3' contains a transfer fault with respect to machine M_3 in state q_2.

Figure 5.27 Testing tree of machine M_3 in Figure 5.26.

Example 5.39 Figure 5.27 shows a testing tree for machine M_3 in Figure 5.26. To obtain the 1-switch cover, we traverse the testing tree from the root and enumerate all complete paths. Each path is represented by the sequence s of input symbols that label the corresponding links in the tree. The sequence s corresponding to each path is concatenated with the input alphabet of M as $s.x$ where $x \in X$, X being the input alphabet. Traversing the testing tree in Figure 5.27 and concatenating as described gives us the following 1-switch cover.

$$T = \{aba, abb, aaa, aab, baa, bab, bba, bbb\}$$

We leave it to you to verify that T is a 1-switch cover for M_3. Note that test $aba \in T$ distinguishes M_3 from M_3' as $O_{M_3}(q_1, abb) = 000 \neq O_{M_3'}(q_1, abb)$. Exercise 5.30 asks for the derivation of a test set from M_2 in Figure 5.25 that is adequate with respect to 1-switch cover criterion.

5.9.2 Comparing automata theoretic methods

Figure 5.28 shows the relative fault detection effectiveness of the transition tour (TT), Unique Input/Output (UIO), UIOv, Distinguishing Sequence (DS) Wp-, and W-methods. As is shown in the figure, the W-method, UIOv, Wp, and DS can detect all faults using the fault model described in Section 5.4.

The distinguishing sequence method constructs an input sequence x from an FSM M such that $O(q_i, x)$ is different for each q_i in M. Sequences so constructed are able to detect all faults in our fault model. The UIO method is able to detect all output faults along transitions but not necessarily all transfer faults.

Figure 5.28 Relative fault detection effectiveness and length of the test sets of various automata theoretic techniques.

Note that the transition tour method has the lowest fault detection capability. In this method, test sequences are generated randomly until all transitions in the underlying FSM are covered. Redundant sequences are removed using a minimization procedure. One reason for the low fault detection ability of the TT method is that it checks only whether or not a transition has been covered and does not check the correctness of the start and tail states of the transition (see Exercise 5.25).

Tests generated using the transition tour method have the least fault detection effectiveness among those generated using other techniques considered in this chapter.

The relative lengths of test sequences is shown in Figure 5.28. Once again the W-method generates the longest, and the largest, set of test sequences while the TT method the shortest. From the relative lengths and fault detection relationships, we observe that larger tests tend to be more powerful than shorter tests. However, we also observe that smaller tests can be as powerful in their fault detection ability as the larger tests as indicated by comparing the fault detection effectiveness of the Wp- with that of the W-method.

In general the W-method generates larger test sequences than the Wp-method. However, the fault detection effectiveness of the tests generated are the same.

5.10 Tools

There are only a few freely available tools for generating tests from finite state models. However, a number of commercial tools are available that enable the tasks of model specification, exploration, and test generation. Some are mentioned below.

Spec Explorer: This tool from Microsoft is a giant in terms of what it offers to a tester. It allows the representation of a meta state model in Spec# and Asml languages that are based on C#. This model is an expression of the program's intended behavior. What a program must do and what it must

not do are specified in the Spec# model. The model need not express *all* of a program's intended behavior. Instead, one may create several models from different points of views. From the model, the tool generates a number of FSMs that sample the program's behavior. Tests are generated from each machine.

Documentation and link: http://research.microsoft.com/en-us/projects/specexplorer/

Spec-Explorer, T-UPPAAL, ModelJUnit are publicly available tools for generating tests from timed automata, FSMs, and EFSMs.

T-UPPAAL: This is a powerful tool for testing embedded real-time systems. The input to T-UPPAAL is a model of the system under test as a timed automata. Using this formal model, T-UPPAAL generates and executes timed test sequences.

Documentation and link: http://people.cs.aau.dk/~marius/tuppaal/

ModelJUnit: This is a Java library that extends JUnit. An FSM or EFSM model is specified as a set of Java classes. Tests are generated from these models.

Documentation and link: http://czt.sourceforge.net/modeljunit/index.html

WMethod: This is an easy-to-use tool to generate tests from an FSM. It takes an FSM as input and generates tests using the W-method described in this chapter.

Documentation and link: http://www.cs.purdue.edu/homes/apm/foundationsBook/

SUMMARY

The area of test generation from finite state models is wide. Research in this area can be categorized as follows:

- test generation techniques,
- empirical studies, and
- test methodology and architectures.

This chapter has focused primarily on test generation techniques from deterministic finite state models. Of the many available, three techniques, namely, the W-method, the UIO method, and

the Wp-method, have been described. These methods are selected for inclusion due to their intrinsic importance in the field of FSM-based test generation and their high fault detection effectiveness. Each of these methods has also found its way into test generation from more complex models such as statecharts and timed automata. Thus, it is important for the reader to be familiar with these foundational test generation methods and prepare for more advanced and complex test generation methods.

Test generation from each of the techniques introduced in this chapter has been automated by several researchers. Nevertheless, students may find it challenging to write their own test generators and conduct empirical studies on practical FSM models.

Exercises

5.1 Modify the DIGDEC machine shown in Figure 5.3 so that it accepts a string over the alphabet $X = \{d, *\}$. For example, strings such as $s = 324*41*9*199**230*$ are valid inputs. An asterisk causes the machine to execute the OUT(num) function where num denotes the decimal number corresponding to the digits received prior to the current asterisk but after any previous asterisk. Thus, the output of the machine for input string s will be

OUT(324) OUT(41) OUT(9) OUT(199) OUT(230)

Notice that the machine does not perform any action if it receives an asterisk following another asterisk.

5.2 Show that (a) *V-equivalence* and *equivalence* of states and machines as defined in Section 5.2.3 are equivalence relations, i.e. they obey the reflexive, symmetric, and transitive laws; (b) if two states are k-distinguishable for any $k > 0$, then they are also distinguishable for any $n \geq k$.

5.3 Show that it is not necessary for equivalent machines to have an equal number of states.

5.4 Show that the set W of Example 5.7 is a characterization set for the machine in Figure 5.13.

5.5 Prove that the FSM M must be a *minimal* machine for the existence of a characterization set.

5.6 Prove that the method for the construction of k-equivalence partitions described in Example 5.8 will always converge, i.e. there will be a table P_n, $n > 0$, such that $P_n = P_{n+1}$.

5.7 The W-procedure for the construction of the W-set from a set of k-equivalence partitions is given on page 239. This is a brute force procedure that determines a distinguishing sequence for every pair of states. From Example 5.9, we know that two or more pairs of states might be distinguishable by the same input sequence. Rewrite the W-procedure by making use of this fact.

5.8 Prove that the k-equivalence partition of a machine is unique.

5.9 Prove that in an FSM with n states, at most $n-1$ constructions of the equivalence classes are needed, i.e. one needs to construct only $P_1, P_2, \ldots, P_{n-1}$.

5.10 Given the FSM of Example 5.6, construct all mutants that can be obtained by adding a state to M.

5.11 Generate a set T of input sequences that distinguish all mutants in Figure 5.12 from machine M.

5.12 Show that any *extra* or *missing* state error in the implementation of design M will be detected by at least one test generated using the W-method.

5.13 Construct the characterization set W and the transition cover for the machine in Figure 5.23(a). Using the W-method, construct set Z assuming that $m = 3$ and derive a test set T. Does any element of T reveal the transfer error in the IUT of Figure 5.23(b)? Compare T with the test set in Example 5.37 with respect to the number of tests and the average size of test sequences.

5.14 Consider the design specification M as shown in Figure 5.15(a). Further, consider an implementation M_3 of M as shown in Figure 5.29. Find all tests in T_1 and T_2 from Example 5.17 that reveal the error in M_3.

Figure 5.29 An implementation of M shown Figure 5.15(a) with a transfer error in state q_1 on input a.

5.15 Consider the design specification in Figure 5.30(a). It contains three states q_0, q_1, and q_2. The input alphabet $X = \{a, b, c\}$, and the output alphabet is $Y = \{0, 1\}$. (a) Derive a transition cover set P, the state cover set S, the characterization set W, and state identification sets for each of the three states. (b) From M, derive a test set T_w using the W-method and test set T_{wp} using the Wp-method. Compare the sizes of the two sets. (c) Figure 5.30(b) shows the transition diagram of an implementation of M that contains a transfer error in state q_2. Which tests in T_w and T_{wp} reveal this error? (d) Figure 5.30(c) shows the transition diagram of an implementation of M that contains an extra state q_3, and a transfer error in state q_2. Which tests in T_w and T_{wp} reveal this error?

Figure 5.30 Three FSMs. (a) Design specification machine, (b) and (c) are machine indicating an erroneous implementation of the FSM in (a).

5.16 (a) Given an FSM $M = (X, Y, Q, q_0, \delta, O)$ where $|X| = n_x$, $|Y| = n_y$, $|Q| = n_z$, calculate the upper bound on the number of tests that will be generated using the W-method. (b) Under what condition(s) will the number of tests generated by the Wp-method be the same as those generated by the W-method?

5.17 Using the tests generated in Example 5.18, determine at least one test for each of the two machines in Figure 5.16 that will reveal the error. Is it necessary to apply phase 2 testing to reveal the error in each of the two machines?

5.18 What is the difference between sequences in the W set and the UIO sequences? Find UIO sequences for each state in the machine with transition diagram shown in Figure 5.13. Find the distinguishing signature for states for which a UIO sequence does not exist.

5.19 What will be the value of counter k when control arrives at Step 2.3 in procedure *gen-long-uio*?

5.20 For machine M_2 used in Example 5.32, what output sequences are generated when the input portion of $Sig(q_1)$ is applied to states q_2 and q_3?

5.21 Suppose that tests $TE(e_1)$ and $TE(e_2)$ are derived using the method in Section 5.8.7 for a given FSM M. Is it possible that $TE(e_1) = TE(e_2)$?

5.22 Generate UIO sequences for all states in the specification shown in Figure 5.13. Using the sequences so generated, develop tests for weak conformance testing of an IUT built that must behave as per the specification given.

5.23 Generate tests for weak conformance testing of machine M_2 in Figure 5.18. Use the UIO sequences for M_2 given on page 259.

5.24 Consider an implementation of the machine shown in Figure 5.21. The state diagram of the implementation is shown in Figure 5.31. Note that the IUT has two errors, a transfer error in state q_2 and an operation error in state q_6. In Example 5.36, the transfer error in q_2 is revealed by test 4 (and 13). Are any of these tests successful in revealing the error in q_2? Is there any test derived earlier that will reveal the error in q_2? Which one of the tests derived in Example 5.33 and 5.35 will reveal the error in q_6?

Figure 5.31 State diagram of an erroneous implementation of the machine in Figure 5.21.

5.25 Transition tours is a technique to generate tests from specification FSMs. A transition tour is an input sequence that when applied to an FSM in its start state traverses each edge at least once. (a) Find a transition tour for each of the two FSMs shown in Figure 5.18. (b) Show that a transition tour is able to detect all operation errors but may not be able to detect all transfer errors. Assume that the FSM specification satisfies the assumptions in Section 5.6.1. (c) Compare the size of tests generated using transition tours and the W-method.

5.26 In Example 5.38, we developed adequate tests to show that certain errors in the IUT are not detected. Derive tests using the W-method, and then using the Wp-method, and show that each test set derived detects all errors shown in Figure 5.32.

Figure 5.32 A transfer error.

5.27 FSM models considered in this chapter are pure in the sense that they capture only the control flow and ignore data definitions and uses. In this exercise, you will learn how to enhance a test set derived using any of the methods described in this chapter by accounting for data flows.

Figure 5.33 shows machine M, an augmented version of the FSM in Figure 5.13. We assume that the IUT corresponding to the FSM in Figure 5.33 uses a local variable Z. This variable is *defined* along transitions $tr_1 = (q_1, q_4)$ and $tr_3 = (q_3, q_5)$. Variable Z is *used* along transitions $tr_2 = (q_2, q_1)$ and $tr_4 = (q_3, q_1)$. Also, x and y are parameters of, respectively, inputs b and a. A data flow path for variable Z is a sequence Tr of transitions such that Z is defined on one transition along Tr and subsequently used along another transition also in Tr. For example, tr_1, tr_5, tr_4 is a data flow path for Z in M, where Z is defined along tr_1 and used along tr_4. We consider only finite length paths and those with only one definition and one corresponding use along a path.

While testing the IUT against M, we must ensure that all data flow paths are tested. This is to detect faults in the defining or usage transitions. (a) Enumerate all data flow paths for variable Z in M. (b) Derive tests, i.e. input sequences, that will traverse all data flow paths derived in (a). (c) Consider a test set T derived for M using the W-method. Does exercising M against all elements of T exercise all data flow paths derived in (a)? (Note: Chapter 7 provides details of data-flow based assessment of test adequacy and enhancement of tests.)

Figure 5.33 FSM of Figure 5.13 augmented with definition and use of local variable Z.

5.28 Let T be a set of test inputs that forms an n-switch set cover for a minimal n-distinguishable FSM M. Prove that T can detect all transfer, operation, extra, and missing state errors in M.

5.29 Show that a test set T that is adequate with respect to the boundary-interior coverage criterion may not be adequate with respect to the 1-switch cover criterion.

5.30 Derive a test set T from M_2 shown in Figure 5.25 that is adequate with respect to the 1-switch cover criterion. Which test in T distinguishes M_2 from M_2' and thereby reveals the error? Compare T derived in this exercise with the test given in Example 5.38 and identify a property of the method used to construct T that allows T to distinguish M_2 from M_2'.

6

Test Generation from Combinatorial Designs

CONTENTS

6.1 Combinatorial designs
6.2 A combinatorial test design process
6.3 Fault model
6.4 Latin squares
6.5 Mutually orthogonal latin squares
6.6 Pairwise design: binary factors
6.7 Pairwise design: multi-valued factors
6.8 Orthogonal arrays
6.9 Covering and mixed-level covering arrays
6.10 Arrays of strength > 2
6.11 Generating covering arrays
6.12 Tools

The purpose of this chapter is to introduce techniques for the generation of test configurations and test data using the combinatorial design techniques with program inputs and their values as factors and levels respectively. These techniques are useful when testing a variety of applications. They allow selection of a small set of test configurations from an often impractically large set, and are effective in detecting faults arising out of factor interactions.

6.1 Combinatorial Designs

Software applications are often designed to work in a variety of environments. Combinations of factors such as the operating system, network connection, and hardware platform, lead to a variety of environments. Each environment corresponds to a given set of values for each factor, known as a *test configuration*. For example, Windows XP, Dial-up connection, and a PC with 1 GB of main memory, is one possible configuration. To ensure high reliability across the intended environments, the application must be tested under as many test configurations, or environments, as possible. However, as illustrated in examples later in this chapter, the number of such test configurations could be exorbitantly large making it impossible to test the application exhaustively.

An analogous situation arises in the testing of programs that have one or more input variables. Each test run of a program often requires at least one value for each variable. For example, a program to find the greatest common divisor of two integers x and y requires two values, one corresponding to x and the other to y. In earlier chapters, we have seen how program inputs can be selected using techniques such as equivalence partitioning and boundary value analysis. While these techniques offer a set of guidelines to design test cases, they suffer from two shortcomings: (a) they raise the possibility of a large number of subdomains in the partition of the input space and (b) they lack guidelines on how to select inputs from various subdomains in the partition.

The number of subdomains in a partition of the input domain increases in direct proportion to the number and type of input variables, and especially so when multidimensional partitioning is used. Also, once a partition is determined, one selects at random a value from each of the subdomains. Such a selection procedure, especially when using unidimensional equivalence partitioning, does not account for the possibility of faults in the program under test that arise due to specific interactions amongst values of different input variables. While boundary value analysis leads to the selection of test cases that test a program at the boundaries of the input domain, other interactions in the input domain might remain untested.

This chapter describes several techniques for generating test configurations or test sets that are small even when the set of possible configurations, or the input domain, and the number of subdomains in its partition, is large and complex. The number of test configurations, or the test set so generated, has been found to be effective in the discovery of faults due to the interaction of various input variables. The techniques we describe here are known by several names such as design of experiments, combinatorial designs, orthogonal designs, interaction testing, and pairwise testing.

6.1.1 Test configuration and test set

In this chapter, we use the terms *test configuration* and *test set* interchangeably. Even though we use the terms interchangeably, they do have different meaning in the context of software testing. However, as the techniques described in this chapter apply to the generation of both test configurations as well as test sets, we have taken the liberty of using them interchangeably. One must be aware that a test configuration is usually a static selection of factors, such as the hardware platform or an operating system. Such selection is completed prior to the start of the test. In contrast, a test set is a collection of test cases used as input during the test process.

6.1.2 Modeling the input and configuration spaces

The input space of a program P consists of k-tuples of values that could be input to P during execution. The configuration space of P consists of all possible settings of the environment variables under which P could be used.

> **Example 6.1** Consider program P that takes two integers $x > 0$ and $y > 0$ as inputs. The input space of P is the set of all pairs of positive non-zero integers. Now suppose that this program is intended to be executed under the Windows and the Mac OS operating system, through the Firefox or Safari browsers, and must be able to print to a local or a networked printer. The configuration space of P consists of triples (X, Y, Z), where X represents an operating system, Y a browser, and Z a local or a networked printer.

Next, consider a program P that takes n inputs corresponding to variables $X_1, X_2, \ldots, X_n$. We refer to the inputs as *factors*. The inputs are also referred to as *test parameters* or as *values*. Let us assume that each factor may be set at any one from a total of $c_i, 1 \leq i \leq n$ values. Each value assignable to a factor is known as a *level*. The notation $|F|$ refers to the number of levels for factor F.

The environment under which an application is intended to be used, generally contributes one or more factors. In Example 6.1, the operating system, browser, and printer connection are three factors that will likely affect the operation and performance of P.

A set of values, one for each factor, is known as a *factor combination*. For example, suppose that program P has two input variables X and Y. Let us say that during an execution of P, X and Y may each assume a value from the set $\{a, b, c\}$ and $\{d, e, f\}$, respectively. Thus we have two factors and three levels for each factor. This leads to a total of $3^2 = 9$ factor combinations, namely $(a, d), (a, e), (a, f), (b, d), (b, e), (b, f), (c, d), (c, e)$, and (c, f).

In general, for k factors with each factor assuming a value from a set of n values, the total factor combinations is n^k.

Suppose now that each factor combination yields one test case. For many programs, the number of tests generated for exhaustive testing could be exorbitantly large. For example, if a program has 15 factors with four levels each, the total number of tests is $4^{15} \approx 10^9$. Executing a billion tests might be impractical for many software applications.

There are special combinatorial design techniques that enable the selection of a small subset of factor combinations from the complete set. This sample is targeted at specific types of faults known as *interaction* faults. Before we describe how the combinatorial designs are generated, let us look at a few examples that illustrate where they are useful.

Example 6.2 Let us model the input space of an online Pizza Delivery Service (PDS) for the purpose of testing. The service takes orders online, checks for their validity, and schedules Pizza for delivery. A customer is required to specify the following four items as part of the online order: Pizza size, Toppings list, Delivery address, and a home phone number. Let us denote these four factors by `S`, `T`, `A`, and `P`, respectively.

Suppose now that there are three varieties for size: `Large`, `Medium`, and `Small`. There is a list of six toppings from which to select. In addition, the customer can customize the toppings. The delivery address consists of customer name, one line of address, city, and the zip code. The phone number is a numeric string possibly containing the dash ("–") separator.

The table below lists one model of the input space for the PDS. Note that while for `Size` we have selected all three possible levels, we have constrained the other factors to a smaller set of levels. Thus, we are concerned with only one of two types of values for `Toppings`, Custom or Preset, and one of the two types of values for factors `Address` and `Phone`, namely `Valid` and `Invalid`.

Factor	**Levels**		
`Size`	Large	Medium	Small
`Toppings`	Custom	Preset	
`Address`	Valid	Invalid	
`Phone`	Valid	Invalid	

The total number of factor combinations is $2^4 + 2^3 = 24$ (alternately, $3 \times 2 \times 2 \times 2 = 24$). However, as an alternate to the table above, we could consider $6 + 1 = 7$ levels for `Toppings`. This would increase

the number of combinations to $2^4 + 5 \times 2^3 + 2^3 + 5 \times 2^2 = 84$ (alternately, $3 \times 7 \times 2 \times 2 = 84$). We could also consider additional types of values for `Address` and `Phone` which would further increase the number of distinct combinations. Notice that if we were to consider each possible valid and invalid string of characters, limited only by length, as a level for `Address`, we will arrive at a huge number of factor combinations.

Later in this section we explain the advantages and disadvantages of limiting the number of factor combinations by partitioning the set of values for each factor into a few subsets. Notice also the similarity of this approach with equivalence partitioning. The next example illustrates factors in a Graphical User Interface (GUI).

Example 6.3 The GUI of application T consists of three menus labeled `File`, `Edit`, and `Typeset`. Each menu contains several items listed below.

Factor	Levels			
`File`	New	Open	Save	Close
`Edit`	Cut	Copy	Paste	Select
`Typeset`	LaTex	BibTeX	Plain TeX	MakeIndex

We have three factors in T. Each of these three factors can be set to any of four levels. Thus, we have a total of $4^3 = 64$ factor combinations.

Note that each factor in this example corresponds to a relatively smaller set of levels when compared to factors `Address` and `Phone` in the previous example. Hence the number of levels for each factor is set equal to the cardinality of the set of the corresponding values.

Example 6.4 Let us now consider the Unix `sort` utility for sorting ASCII data in files or obtained from the standard input. The utility has several options and makes an interesting example for the identification of factors and levels. The command line for `sort` is as given below.

`sort` [-cmu] [-o output] [-T directory] [-y [kmem]] [-z recsz] [-dfiMnr] [- b] [t char] [-k keydef] [+pos1 [-pos2]] [file...]

Tables 6.1 and 6.2 list all the factors of `sort` and their corresponding levels. Note that the levels have been derived using equivalence partitioning for each option and are not unique. We have decided

Table 6.1 Factors and levels for the Unix `sort` utility.

Factor	Meaning	Levels			
–	Forces the source to be the standard input.	Unused	Used		
–c	Verify that the input is sorted according to the options specified on the command line.	Unused	Used		
–m	Merge sorted input	Unused	Used		
–u	Suppress all but one of the matching keys.	Unused	Used		
–o *output*	Output sent to a file.	Unused	Valid file	Invalid file	
–T*directory*	Temporary directory for sorting.	Unused	Exists	Does not exist	
–y*kmem*	Use *kmem* kilobytes of memory for sorting.	Unused	Valid *kmem*	Invalid *kmem*	
–zrecsize	Specifies record size to hold each line from the input file.	Unused	Zero size	Large size	
–d*fiMnr*	Perform dictionary sort.	Unused	fi	Mnr	fiMnr

to limit the number of levels for each factor to 4. You could come up with a different, and possibly a larger or a smaller, set of levels for each factor.

In Tables 6.1 and 6.2, level *Unused* indicates that the corresponding option is not used while testing the `sort` command. *Used* means that the option is used. Level *Valid File* indicates that the file specified using the `-o` option exists whereas *Invalid file* indicates that the specified file does not exist. Other options can be interpreted similarly.

We have identified a total of 20 factors for the `sort` command. The levels listed in Table 6.1 lead to a total of approximately 1.9×10^9 combinations.

Table 6.2 Factors and levels for the Unix `sort` utility (continued).

Factor	Meaning	Levels			
−f	Ignore case.	Unused	Used		
−i	Ignore non-ASCII characters.	Unused	Used		
−M	Fields are compared as months.	Unused	Used		
−n	Sort input numerically.	Unused	Used		
−r	Reverse the output order.	Unused	Used		
−b	Ignore leading blanks when using +pos1 and -pos2.	Unused	Used		
−t*c*	Use character *c* as field separator.	Unused	c_1	c_1c_2	
−k*keydef*	Restricted sort key definition.	Unused	start	end	start*type*
+*pos1*	Start position in the input line for comparing fields.	Unused	f.c	f	0.c
−*pos2*	End position in the input line for comparing fields.	Unused	f.c	f	0.c
file	File to be sorted.	Not specified	Exists	Does not exist	

Example 6.5 There is often a need to test a web application on different platforms to ensure that a claim such as "Application X can be used under Windows and OS X." Here we consider a combination of hardware, operating system, and a browser as a platform. Such testing is commonly referred to as *compatibility* testing.

Let us identify factors and levels needed in the compatibility testing of application X. Given that we would like X to work on a

Table 6.3 Factors and levels for testing Web application X.

Hardware	Operating System	Browser
Dell Dimension Series	Windows Server 2008-Web Edition	Internet Explorer 9
Apple iMac	Windows Server 2008-R2 standard Edition	Internet Explorer
Apple MacBook Pro	Windows 7 Enterprise	Chrome
	OS 10.6	Safari 5.1.6
	OS 10.7	Firefox

variety of hardware, OS, and browser combinations, it is easy to obtain three factors, i.e. hardware, OS, and browser. These are listed in the top row of Table 6.3. Notice that instead of listing factors in different rows, we now list them in different columns. The levels for each factor are listed in rows under the corresponding columns. This has been done to simplify the formatting of the table.

A quick examination of the factors and levels in Table 6.3 reveals that there are 75 factor combinations. However, some of these combinations are infeasible. For example, OS 10.2 is an OS for the Apple computers and not for the Dell Dimension series PCs. Similarly, the Safari browser is used on Apple computers and not on the PC in the Dell Series. While various editions of the Windows OS can be used on an Apple computer using an OS bridge such as the Virtual PC or Boot Camp, we assume that this is not the case for testing application X.

The discussion above leads to a total of 40 infeasible factor combinations corresponding to the hardware–OS combination and the hardware–browser combination. Thus, in all we are left with 35 platforms on which to test X.

Notice that there is a large number of hardware configurations under the Dell Dimension Series. These configurations are obtained by selecting from a variety of processor types, e.g., Pentium versus Athelon, processor speeds, memory sizes, and several others. One could replace the Dell Dimension Series in Table 6.3 by a few selected configurations. While doing so will lead to more thorough testing of application X, it will also increase the number of factor combinations, and hence the time to test.

Identifying factors and levels allows us to divide the input domain into subdomains, one for each factor combination. The design of test cases can

now proceed by selecting at least one test from each subdomain. However, as shown in examples above, the number of subdomains could be too large and hence the need for further reduction. The problem of test case construction and reduction in the number of test cases is considered in the following sections.

6.2 A Combinatorial Test Design Process

Figure 6.1 shows a three-step process for the generation of test cases and test configurations for an application under test. The process begins with a model of the input space if test cases are to be generated. The application environment is modeled when test configurations are to be generated. In either case, the model consists of a set of factors and the corresponding levels. The modeling of input space or the environment is not exclusive and one might apply either one or both depending on the application under test. Examples in the previous section illustrate the modeling process.

Test generation using combinatorial designs can be considered as a three-step process: build a model, generate a combinatorial design, generate tests.

In the second step, the model is input to a combinatorial design procedure to generate a combinatorial object which is simply an array of factors and levels. Such an object is also known as a *factor covering design*. Each row in this array generates at least one test configuration or one test input. In this chapter, we have described several procedures for generating a combinatorial object. The procedures described make use of Latin squares, orthogonal arrays, mixed orthogonal arrays, covering arrays, and mixed-level covering arrays. While all procedures, and their variants introduced in this chapter are used in software testing, covering arrays, and mixed-level covering arrays seem to be the most useful in this context.

A combinatorial object is an array of values, factors and levels. Such an object is also referred to as a factor covering design as the array has at least one occurrence of each value of each factor.

Figure 6.1 A process for generating tests and test configurations using combinatorial designs. A combinatorial design is represented as a combinatorial object which is an array of size $N \times k$ with N rows, one corresponding to at least one test run and k columns one corresponding to each factor.

In the final step, the combinatorial object generated is used to design a test set or a test configuration as the requirement might be. The combinatorial object is an array of factor combinations. Each factor combination may lead to one or more test cases where each test case consists of values of input variables and the expected output. However, all combinations generated might not be feasible. Further, the sequence in which test inputs are entered is also not specified in the combinations. The next few examples illustrate how a factor combination may lead to one or more test cases, both feasible and infeasible and assist in detecting errors.

Of the three steps shown in Figure 6.1, the second and third steps can be automated. There are commercial tools available that automate the second step, i.e. the generation of a combinatorial object. Generation of test configurations and test cases requires a simple mapping from integers to levels or values of factors and input variables, a relatively straightforward task.

Example 6.6 From the factor levels listed in Example 6.3, we can generate 75 test cases, one corresponding to each combination. Thus, for example, the next two test inputs are generated from the table in Example 6.3.

$< t_1 : \texttt{File} = \text{Open}, \texttt{Edit} = \text{Paste}, \texttt{Typeset} = \text{MakeIndex} >$
$< t_2 : \texttt{File} = \text{New}, \texttt{Edit} = \text{Cut}, \texttt{Typeset} = \text{LaTeX} >$

Let us assume that the values of `File`, `Edit`, and `Typeset` are set in the sequence listed. Test t_1 requires that the tester select "Open" from the `File` menu, followed by "Paste" from the `Edit` menu, and finally "MakeIndex" from the `Typeset` menu. While this sequence of test inputs is feasible, i.e. can be applied as desired, the sequence given in t_2 cannot be applied.

To test the GUI using t_2, one needs to open a "New" file and then use the "Cut" operation from the `Edit` menu. However, the "Cut" operation is usually not available when a new file is opened, unless there is an error in the implementation. Thus, while t_2 is infeasible if the GUI is implemented correctly, it becomes feasible when there is an error in the implementation. While one might consider t_2 to be a useless test case, it does provide a useful sequence of input selections from the GUI menus in that it enforces a check for the correctness of certain features.

Example 6.7 Each combination obtained from the levels listed in Table 6.1 can be used to generate many test inputs. For example, consider the combination in which all factors are set to "Unused" except the $-o$ option which is set to "Valid File" and the *file* option that is set to "Exists." Assuming that files "afile," "bfile," "cfile," and "dfile" exist, this factor setting can lead to many test cases, two of which are listed below.

$< t_1 :$ `sort` $-o$afile bfile $>$
$< t_2 :$ `sort` $-o$cfile dfile $>$

One might ask as to why only one of t_1 and t_2 is not sufficient. Indeed, t_2 might differ significantly from t_1 in that *dfile* is a very large sized file relative to *bfile*. Recall that both *bfile* and *dfile* contain data to be sorted. One might want to test `sort` on a very large file to ensure its correctness and also to measure its performance relative to a smaller data file. Thus, in this example, two tests generated from the same combination are designed to check for correctness and performance.

To summarize, a combination of factor levels is used to generate one or more test cases. For each test case, the sequence in which inputs are to be applied to the program under test must be determined by the tester. Further, the factor combinations do not indicate in any way the sequence in which the generated tests are to be applied to the program under test. This sequence too must be determined by the tester. The sequencing of tests generated by most test generation techniques must be determined by the tester and is not a unique characteristic of tests generated in combinatorial testing.

6.3 Fault Model

Tests generated using combinatorial designs aim at uncovering interaction faults. Such faults arise due to an interaction of two or more factors.

The combinatorial design procedures described in this chapter are aimed at generating test inputs and test configurations that might reveal certain types of faults in the application under test. We refer to such faults as *interaction* faults. We say that an interaction fault is *triggered* when a certain combination of $t \geq 1$ input values causes the program containing the fault to enter an invalid state. Of course, this invalid state must propagate to a point in the program execution where it is observable and hence is said to *reveal* the fault.

Faults triggered by some value of an input variable, i.e. $t = 1$, regardless of the values of other input variables, are known as *simple* faults. For $t = 2$,

A pairwise interaction fault occurs due to an interaction of two factors. A t-way interaction fault occurs due to the interaction of t factors. When $t = 1$, the fault is considered simple.

they are known as pairwise interaction faults, and in general, for any arbitrary value of t, as t-way interaction faults. An t–way interaction fault is also known as a t-factor fault. For example, a pairwise interaction fault will be triggered only when two input variables are set to specific values. A 3-way interaction fault will be triggered only when three input variables assume specific values. The next two examples illustrate interaction faults.

Example 6.8 Consider the following program that requires three inputs x, y, and z. Prior to program execution, variable x is assigned a value from the set $\{x_1, x_2, x_3\}$, variable y a value from the set $\{y_1, y_2, y_3\}$, and variable z a value from the set $\{z_1, z_2\}$. The program outputs the function $f(x, y, z)$ when $x = x_1$ and $y = y_2$, function $g(x, y)$ when $x = x_2$ and $y = y_1$, and function $f(x, y, z) + g(x, y)$, otherwise.

Program P6.1

```
begin
  int x, y, z;
  input (x, y, z);
  if(x==$x_1$ and y==$y_2$)
    output (f(x, y, z));
  else if(x==$x_2$ and y==$y_1$)
   output (g(x, y));
  else
   output (f(x, y, z)+g(x, y))   // This statement
end                                is not protected correctly.
```

As marked, Program P6.1 contains one error. The program must output $f(x, y, z) - g(x, y)$ when $x = x_1$ and $y = y_1$ and $f(x, y, z) + g(x, y)$ when $x = x_2$ and $y = y_2$. This error will be revealed by executing the program with $x = x_1, y = y_1$, and any value of z, if $f(x_1, y_1, *) - g(x_1, y_1) \neq f(x_1, y_1, *) + g(x_1, y_1)$. This error is an example of a pairwise interaction fault revealed only when inputs x and y interact in a certain way. Notice that variable z does not play any role in triggering the fault but it might need to take a specific value for the fault to be revealed (also see Exercises 6.2 and 6.3).

Example 6.9 A missing condition is the cause of a pairwise interaction fault in Program P6.1. Also, the input variables are compared against their respective values in the conditions governing the flow of control. Program P6.2 contains a 3-way interaction fault in

which an incorrect arithmetic function of three input variables is responsible for the fault.

Let the three variables assume input values from their respective domains as follows: $x \in \{-1, 1\}$, $y \in \{-1, 0\}$, and $z \in \{0, 1\}$. Notice that there is a total of eight combinations of the three input variables.

```
Program P6.2
1   begin
2     int x, y, z, p;
3     input (x, y, z);
4     p=(x+y)*z;   // This statement must be
        p=(x-y)*z.
5     if(p≥0)
6     output (f(x, y, z));
7     else
8     output (g(x, y));
9   end
```

The above program contains a 3-way interaction fault. This fault is triggered by all inputs such that $x + y \neq x - y$ and $z \neq 0$ because for such inputs the program computes an incorrect value of p thereby entering an incorrect state. However, the fault is revealed only by the following two of the eight possible input combinations: $x = 1$, $y = -1, z = 1$ and $x = -1, y = -1, z = 1$.

6.3.1 Fault vectors

As mentioned earlier, a t-way fault is triggered whenever a subset of $t \leq k$ factors, out of a complete set of k factors, is set to some set of levels. Given a set of k factors $f_1, f_2, \ldots, f_k$, each at $q_i, 1 \leq i \leq k$ levels, a vector V of factor levels is $(l_1, l_2, \ldots, l_k)$, where $l_i, 1 \leq i \leq k$ is a specific level for the corresponding factor. V is also known as a *run*.

We say that a run V is a *fault vector* for program P if the execution of P against a test case derived from V triggers a fault in P. V is considered as a t-way fault vector if any $t \leq k$ elements in V are needed to trigger a fault in P. Note that a t-way fault vector for P triggers a t-way fault in P. Given k factors, there are $k - t$ don't care entries in a t-way fault vector. We indicate don't care entries with an asterisk. For example, the 2-way fault vector $(2, 3, *)$ indicates that the 2-way interaction fault is triggered when factor 1 is set to level 2, factor 2 to level 3, and factor 3 is the don't care factor.

A vector consisting of specific values, or levels, of each factor is known as a run. A run is considered a fault vector if a test derived from it triggers a fault in the IUT.

A don't care entry in a fault vector is indicated by an asterisk (*). It's value can be set arbitrarily to a factor level and does not affect the fault discovery.

Example 6.10 The input domain of Program P6.2 consists of three factors x, y, and z each having two levels. There is a total of eight runs. For example, $(1, 1, 1)$ and $(-1, -1, 0)$ are two runs. Of these eight runs, $(-1, 1, 1)$ and $(-1, -1, 1)$ are fault vectors that trigger the 3-way fault in Program P6.2. $(x_1, y_1, *)$ is a 2-way fault vector given that the values x_1 and y_1 trigger the 2-way fault in Program P6.2.

The aim of the test generation techniques described in this chapter is to generate tests that discover t-way faults. The number of tests generated increase with t.

The goal of the test generation techniques described in this chapter is to generate a sufficient number of runs such that tests generated from these runs reveal all t-way faults in the program under test. As we see later in this chapter, the number of such runs increases with the value of t. In many practical situations, t is set to 2 and hence the tests generated are expected to reveal pairwise interaction faults. Of course, while generating t-way runs, one also generates some $t+1, t+2, \ldots, t+k-1$, and k-way runs. Hence, there is always a chance that runs generated with $t = 2$ reveal some higher level interaction faults.

6.4 Latin Squares

In the previous sections, we have shown how to identify factors and levels in a software application and how to generate test cases from factor combinations. Considering that the number of factor combinations could be exorbitantly large, we want to examine techniques for test generation that focus on only a certain subset of factor combinations.

Latin squares and mutually orthogonal Latin squares are rather ancient devices found useful in selecting a subset of factor combinations from the complete set. In this section, we introduce Latin squares that are used to generate fewer factor combinations than what would be generated by the brute force technique mentioned earlier. Mutually orthogonal Latin squares and the generation of a small set of factor combinations are explained in the subsequent sections.

A Latin square of order n is an $n \times n$ matrix of n-symbols such that no symbol appears more than once in a row or a column.

Let S be a finite set of n symbols. A Latin square of order n is an $n \times n$ matrix such that no symbol appears more than once in a row and a column. The term "Latin square" arises from the fact that the early versions used letters from the Latin alphabet A, B, C, and so on in a square arrangement.

Example 6.11 Given $S = \{A, B\}$, here are two Latin squares of order 2.

A	B		B	A
B	A		A	B

Given $S = \{1, 2, 3\}$, we have the following three Latin Squares of order 3.

1	2	3	2	3	1	2	1	3
2	3	1	1	2	3	3	2	1
3	1	2	3	1	2	1	3	2

Additional Latin squares of order 3 can be constructed by permuting rows, columns, and by exchanging symbols, e.g. by replacing all occurrences of symbol 2 by 3 and 3 by 2, of an existing Latin square.

Larger Latin squares of order n can be constructed by creating a row of n distinct symbols. Additional rows can be created by permuting the first row. For example, here is a Latin square M of order 4 constructed by cyclically rotating the first row and placing successive rotations in subsequent rows.

Latin squares of a given order n may not be unique. Two Latin squares of order n are considered isomorphic if one can be obtained from the other by a permutation of rows and/or columns and by symbol exchange.

1	2	3	4
2	3	4	1
3	4	1	2
4	1	2	3

Given a Latin square M, a large number of Latin squares of order 4 can be obtained from M through row and column interchange and symbol renaming operations. Two Latin squares M_1 and M_2 are considered *isomorphic* if one can be obtained from the other by permutation of rows, columns, and symbol exchange. However, given a Latin square of order n, not all Latin squares of order n can be constructed using the interchange and symbol renaming.

Example 6.12 Consider the following Latin square M_1 of order 4.

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

M_1 cannot be obtained from M listed earlier by permutation of rows, columns, or symbol exchange. Exercise 6.6 suggests how M_1 can be constructed. Notice that M_1 has three 2×2 sub-tables that are Latin squares containing the symbol 1. However, there are no such 2×2 Latin squares in M.

A Latin square of order $n > 2$ can also be constructed easily by doing modulo arithmetic. For example, the Latin square M of order 4 given below is constructed such that $M(i, j) = i + j (\text{mod } 4), 1 \leq i \leq 4, 1 \leq j \leq 4$.

	1	2	3	4
1	2	3	0	1
2	3	0	1	2
3	0	1	2	3
4	1	2	3	0

A Latin square is considered to be in standard form if the elements in the top row and the leftmost column are arranged in order.

A Latin square based on integers $0, 1, \ldots, n$ is said to be in *standard form* if the elements in the top row and the leftmost column are arranged in order. In a Latin square based on letters A, B, and so on, the elements in the top row and the leftmost column are arranged in alphabetical order.

6.5 Mutually Orthogonal Latin Squares

Two Latin squares are considered mutually orthogonal if the matrix obtained by juxtaposing the two has unique elements. Such Latin squares are also referred to as MOLS.

Let M_1 and M_2 be two Latin squares, each of order n. Let $M_1(i,j)$ and $M_2(i,j)$ denote, respectively, the elements in the ith row and j^{th} column of M_1 and M_2. We now create an $n \times n$ matrix M from M_1 and M_2 such that $M(i,j)$ is $M_1(i,j)M_2(i,j)$, i.e. we simply juxtapose the corresponding elements of M_1 and M_2. If each element of M is unique, i.e. it appears exactly once in M, then M_1 and M_2 are said to be *mutually orthogonal* Latin squares of order n.

Example 6.13 There are no mutually orthogonal Latin squares of order 2. Listed below are two mutually orthogonal Latin squares of order 3.

$$M_1 = \begin{matrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{matrix} \qquad M_2 = \begin{matrix} 2 & 3 & 1 \\ 1 & 2 & 3 \\ 3 & 1 & 2 \end{matrix}$$

To check if indeed M_1 and M_2 are mutually orthogonal, we juxtapose the corresponding elements to obtain the following matrix.

$$L = \begin{matrix} 1\,2 & 2\,3 & 3\,1 \\ 2\,1 & 3\,2 & 1\,3 \\ 3\,3 & 1\,1 & 2\,2 \end{matrix}$$

As each element of M appears exactly once, M_1 and M_2 are indeed mutually orthogonal.

Mutually orthogonal Latin squares are commonly referred to as MOLS. The term MOLS(n) refers to the set of MOLS of order n. It is known that when n is prime, or a power of prime, MOLS(n) contains $n-1$ mutually orthogonal Latin squares. Such a set of MOLS is known as a *complete* set.

MOLS do not exist for $n = 2$ and $n = 6$ but they do exist for all other values of $n > 2$. Numbers 2 and 6 are known as *Eulerian numbers* after the famous mathematician Leonhard Euler (1707–1783). The number of MOLS of order n is denoted by $N(n)$. Thus, when n is prime or a power of prime, $N(n) = n - 1$ (also see Exercise 6.9). The next example illustrates a simple procedure to construct MOLS(n) when n is prime.

There are exactly $n - 1$ MOLS when n is prime or a power of prime. There are no MOLS for $n = 2$ and $n = 6$.

Example 6.14 We follow a simple procedure to construct MOLS(5). We begin by constructing a Latin square of order 5 given the symbol set $S = \{1, 2, 3, 4, 5\}$. This can be done by the method described in the previous section, which is to generate subsequent rows of the matrix by rotating the previous row to the left by one position. The first row is a simple enumeration of all symbols in S; the order of enumeration does not matter. Here is one element of MOLS(5).

$$M_1 = \begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 1 \\ 3 & 4 & 5 & 1 & 2 \\ 4 & 5 & 1 & 2 & 3 \\ 5 & 1 & 2 & 3 & 4 \end{matrix}$$

Next, we obtain M_2 by rotating rows 2 through 5 of M_1 by two positions to the left.

$$M_2 = \begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 1 & 2 \\ 5 & 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 & 1 \\ 4 & 5 & 1 & 2 & 3 \end{matrix}$$

M_3 and M_4 are obtained similarly but by rotating the first row of M_1 by three and four positions, respectively.

$$M_3 = \begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 1 & 2 & 3 \\ 2 & 3 & 4 & 5 & 1 \\ 5 & 1 & 2 & 3 & 4 \\ 3 & 4 & 5 & 1 & 2 \end{matrix}$$

$$M_4 = \begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 1 & 2 & 3 & 4 \\ 4 & 5 & 1 & 2 & 3 \\ 3 & 4 & 5 & 1 & 2 \\ 2 & 3 & 4 & 5 & 1 \end{matrix}$$

Thus, we get MOLS(5) $= \{M_1, M_2, M_3, M_4\}$. It is easy to check that indeed the elements of MOLS(5) are mutually orthogonal by superimposing them pairwise. For example, superimposing M_2 and M_4 leads to the following matrix where each element occurs exactly once.

11	22	33	44	55
35	41	52	13	24
54	15	21	32	43
23	34	45	51	12
42	53	14	25	31

The maximum number of MOLS(n) is $n - 1$ when n is not prime.

The method illustrated in the previous example is guaranteed to work only when constructing MOLS(n) for n that is prime or a power of prime. For other values of n, the maximum size of MOLS(n) is $n - 1$. However, different methods are used to construct MOLS(n) when n is not a prime or a power of prime. Such methods are beyond the scope of this book. There is no general method available to construct the largest possible MOLS(n) for n that is not a prime or a power of prime. The CRC Handbook of Combinatorial Designs (see the Bibliographic Notes section) lists a large set of MOLS for various values of n.

6.6 Pairwise Design: Binary Factors

A factor is considered binary if it can assume one of the two possible values.

We now introduce the techniques for selecting a subset of factor combinations from the complete set. First, we focus on programs whose configuration or input space can be modeled as a collection of factors where each factor assumes one or two values. We are interested in selecting a subset from the complete set of factor combinations such that all pairs of factor levels are covered in the selected subset.

Each factor combination selected will generate at least one test input or test configuration for the program under test. As discussed earlier, the complete set of factor combinations might be too large and impractical to drive a test process, and hence the need for selecting a subset.

To illustrate the process of selecting a subset of combinations from the complete set, suppose that a program to be tested requires three inputs, one corresponding to each input variable. Each variable can take only one of two distinct values. Considering each input variable as a factor, the total number of factor combinations is 2^3. Let X, Y, and Z denote the three input variables and $\{X_1, X_2\}$, $\{Y_1, Y_2\}$, and $\{Z_1, Z_2\}$, their

respective sets of values. All possible combinations of these three factors are given below.

(X_1, Y_1, Z_1) (X_1, Y_1, Z_2)
(X_1, Y_2, Z_1) (X_1, Y_2, Z_2)
(X_2, Y_1, Z_1) (X_2, Y_1, Z_2)
(X_2, Y_2, Z_1) (X_2, Y_2, Z_2)

However, we are interested in generating tests such that each pair of input values is covered, i.e. each pair of input values appears in at least one test. There are 12 such pairs, namely (X_1, Y_1), (X_1, Y_2), (X_1, Z_1), (X_1, Z_2), (X_2, Y_1), (X_2, Y_2), (X_2, Z_1), (X_2, Z_2), (Y_1, Z_1), (Y_1, Z_2), (Y_2, Z_1), and (Y_2, Z_2). In this case, the following set of four combinations suffices.

The number of tests needed to cover all binary factors is significantly less than what is obtained using exhaustive enumeration.

(X_1, Y_1, Z_2) (X_1, Y_2, Z_1)
(X_2, Y_1, Z_1) (X_2, Y_2, Z_2)

The above set of four combinations is also known as a *pairwise design*. Further, it is a *balanced* design because each value occurs exactly the same number of times. There are several sets of four combinations that cover all 12 pairs (see Exercise 6.8).

Notice that being content with pairwise coverage of the input values reduces the required number of tests from eight to four, a 50% reduction. The large number of combinations resulting from 3-way, 4-way, and higher order designs often force us to be content with pairwise coverage.

Let us now generalize the problem of pairwise design to $n \geq 2$ factors where each factor takes on one of two values. To do so, we define S_{2k-1} to be the set of all binary strings of length $2k-1$ such that each string has exactly k 1s. There are exactly $\binom{2k-1}{k}$ strings in S_{2k-1}. Recall that $\binom{n}{k} = \frac{!n}{!(n-k)\ !k}$.

Example 6.15 For $k = 2$, S_3 contains $\binom{3}{2} = 3$ binary strings of length 3, each containing exactly two 1s. All strings are laid out in the following with column number indicating the position within a string and row number indicating the string count.

	1	2	3
1	0	1	1
2	1	0	1
3	1	1	0

For $k = 3$, S_5 contains $\binom{5}{3} = 10$ binary strings of length 5, each containing three 1s.

	1	2	3	4	5
1	0	0	1	1	1
2	0	1	1	1	0
3	1	1	1	0	0
4	1	0	1	1	0
5	0	1	1	0	1
6	1	1	0	1	0
7	1	0	1	0	1
8	0	1	0	1	1
9	1	1	0	0	1
10	1	0	0	1	1

Given the number of two-valued parameters, the following procedure can be used to generate a pairwise design. We have named the procedure as the "SAMNA procedure" after its authors (see the Bibliographic Notes section for details).

`Procedure for generating pairwise designs.`

Input: n: Number of two-valued input variables (or factors).
Output: A set of factor combinations such that all pairs of input values are covered.
Procedure: `SAMNA`

/* $X_1, X_2, \ldots, X_n$ denote the n input variables.
X_i^* denotes one of the two possible values of variable X_i, for $1 \leq i \leq n$.
One of the two values of each variable corresponds to a 0 and the other to 1.
*/

Step 1 Compute the smallest integer k such that $n \leq |S_{2k-1}|$.
Step 2 Select any subset of n strings from S_{2k-1}. Arrange these to form a $n \times (2k-1)$ matrix with one string in each row while the columns contain different bits in each string.
Step 3 Append a column of 0s to the end of each string selected. This will increase the size of each string from $2k-1$ to $2k$.
Step 4 Each one of the $2k$ columns contains a bit pattern from which we generate a combination of factor levels as follows. Each combination is of the kind $(X_1^*, X_2^*, \ldots, X_n^*)$, where the value of each variable is selected depending on whether the bit in column i, $1 \leq i \leq n$ is a 0 or a 1.

`End of procedure SAMNA`

Example 6.16 Consider a simple Java applet named `ChemFun` that allows a user to create an in-memory database of chemical elements and search for an element. The applet has five inputs listed below with their possible values. We refer to the inputs as factors. For simplicity we have assumed that each input has exactly two possible values.

Factor	Name	Levels	Comments
1	Operation	{Create, Show}	Two buttons
2	Name	{Empty, Non-empty}	Data field, string
3	Symbol	{Empty, Non-empty}	Data field, string
4	Atomic number	{Invalid, Valid}	Data field, data > 0
5	Properties	{Empty, Non-empty}	Data field, string

The applet offers two operations via buttons labeled `Create` and `Show`. Pressing the `Create` button causes the element related data, e.g. its name and atomic number, to be recorded in the database. However, the applet is required to perform simple checks prior to saving the data. The user is notified if the checks fail and data entered is not saved. The `Show` operation searches for data that matches information typed in any of the four data fields.

Notice that each of the data fields could take on an impractically large number of values. For example, a user could type almost any string using characters on the keyboard in the `Atomic number` field. However, using equivalence class partitioning, we have reduced the number of data values for which we want to test `ChemFun`.

Testing `ChemFun` on all possible parameter combinations would require a total of $2^5 = 32$ tests. However, if we are interested in testing against all pairwise combinations of the five parameters, then only six tests are required. We now show how these six tests are derived using procedure SAMNA.

Given five binary factors (or parameters), exhaustive testing would require 32 tests. However, pairwise testing requires only six tests.

Input: $n = 5$ factors.
Output: A set of factor combinations such that all pairs of input values are covered.

Step 1 Compute the smallest integer k such that $n \leq |S_{2k-1}|$. In this case we obtain $k = 3$

Step 2 Select any subset of n strings from S_{2k-1}. Arrange these to form a $n \times (2k-1)$ matrix with one string in each row while the columns contain bits from each string.

We select first five of the 10 strings in S_5 listed in Example 6.15. Arranging them as desired gives us the following matrix.

	1	2	3	4	5
1	0	0	1	1	1
2	0	1	1	1	0
3	1	1	1	0	0
4	1	0	1	1	0
5	0	1	1	0	1

Step 3 Append 0s to the end of each selected string. This will increase the size of each string from $2k-1$ to $2k$.

Appending a column of 0s to the matrix above leads to the following 5×6 matrix.

	1	2	3	4	5	6
1	0	0	1	1	1	0
2	0	1	1	1	0	0
3	1	1	1	0	0	0
4	1	0	1	1	0	0
5	0	1	1	0	1	0

Step 4 Each of the $2k$ columns contains a bit pattern from which we generate a combination of factor values as follows. Each combination is of the kind $(X_1^*, X_2^*, \ldots, X_n^*)$, where the value of each variable is selected depending on whether the bit in column $i, 1 \leq i \leq n$, is a 0 or a 1.

The following factor combinations are obtained by replacing the 0s and 1s in each column of the matrix listed above by the corresponding values of each factor. Here we have assumed that the first of two factor levels listed earlier corresponds to a 0 and the second level corresponds to a 1.

	1	2	3	4	5	6
1	Create	Create	Show	Show	Show	Create
2	Empty	NE	NE	NE	Empty	Empty
3	Non-empty	NE	NE	Empty	Empty	Empty
4	Valid	Invalid	Valid	Valid	Invalid	Invalid
5	Empty	NE	NE	Empty	NE	Empty

NE: Non-empty.

The matrix above lists six factor combinations, one corresponding to each column. Each combination can now be used to generate one or more tests as explained in Section 6.2. The following is a set of six tests for the `ChemFun` applet.

Each factor combination leads to one test.

$$
\begin{aligned}
T \quad = \{ \quad & t_1 : < \quad \text{Button} = \text{Create}, \text{Name} = \text{“”}, \text{Symbol} = \text{“C”}, \\
& \qquad\quad \text{Atomic number} = 6, \text{Properties} = \text{“”} > \\
& t_2 : < \quad \text{Button} = \text{Create}, \text{Name} = \text{“Carbon”}, \text{Symbol} = \text{“C”}, \\
& \qquad\quad \text{Atomic number} = -6, \text{Properties} = \text{“Non-metal”} > \\
& t_3 : < \quad \text{Button} = \text{Show}, \text{Name} = \text{“Hydrogen”}, \text{Symbol} = \text{“C”}, \\
& \qquad\quad \text{Atomic number} = 1, \text{Properties} = \text{“Non-metal”} > \\
& t_4 : < \quad \text{Button} = \text{Show}, \text{Name} = \text{“Carbon”}, \text{Symbol} = \text{“C”}, \\
& \qquad\quad \text{Atomic number} = 6, \text{Properties} = \text{“”} > \\
& t_5 : < \quad \text{Button} = \text{Show}, \text{Name} = \text{“”}, \text{Symbol} = \text{“”}, \\
& \qquad\quad \text{Atomic number} = -6, \text{Properties} = \text{“Non-metal”} > \\
& t_6 : < \quad \text{Button} = \text{Create}, \text{Name} = \text{“}'\text{”}, \text{Symbol} = \text{“”}, \\
& \qquad\quad \text{Atomic number} = -6, \text{Properties} = \text{“”} > \\
\}
\end{aligned}
$$

The 5×6 array of 0s and 1s in Step 2 of Procedure SAMNA, is sometimes referred to as a *combinatorial object*. While Procedure SAMNA is one method for generating combinatorial objects, several other methods appear in the following sections.

Each method for generating combinatorial objects has its advantages and disadvantages that are often measured in terms of the total number of test cases, or test configurations, generated. In the context of software testing, we are interested in procedures that allow the generation of a small set of test cases or test configurations, while covering all interaction tuples. Usually, we are interested in covering interaction pairs, though in some cases interaction triples or even quadruples might be of interest.

In software testing, we are interested in generating a small set of tests that will likely reveal all t-way interaction faults.

6.7 Pairwise Design: Multi-Valued Factors

Many practical test configurations are constructed from one or more factors where each factor assumes a value from a set containing more than two values. In this section, we show how to use MOLS to construct test configurations when (a) the number of factors is two or more, (b) the number of levels for each factor is more than two, and (c) all factors have the same number of levels.

Next, we list a procedure for generating test configurations in situations where each factor can be at more than two levels. The procedure uses MOLS(n) when the test configuration requires n factors.

```
Procedure for generating pairwise designs using
mutually orthogonal Latin squares.
```

Input: n: Number of factors.
Output: A set of factor combinations such that all level pairs are covered.
Procedure: PDMOLS

/* $F'_1, F'_2, \ldots, F'_n$ denote the n factors.
$X^*_{i,j}$ denotes the jth level of the ith factor.
*/

Step 1 Relabel the factors as $F_1, F_2, \ldots, F_n$ such that the following ordering constraint is satisfied: $|F_1| \geq |F_2| \geq, \ldots, \geq |F_{n-1}| \geq |F_n|$. Let $b = |F1|$ and $k = |F2|$. Note that two or more labeling are possible when two or more pairs of factors have the same number of levels.

Step 2 Prepare a table containing n columns and $b \times k$ rows divided into b blocks. Label the columns as $F_1, F_2, \ldots, F_n$. Each block contains k rows. A sample table for $n = b = k = 3$, when all factors have an equal number of labels, is shown below.

Block	Row	F_1	F_2	F_3
1	1			
	2			
	3			
2	1			
	2			
	3			
3	1			
	2			
	3			

Step 3 Fill column F_1 with 1s in Block 1, 2s in Block 2, and so on. Fill Block 1 of column F_2 with the sequence $1, 2, \ldots, k$ in rows 1 through k. Repeat this for the remaining blocks. A sample table with columns 1 and 2 filled is shown below.

Block	Row	F_1	F_2	F_3
1	1	1	1	
	2	1	2	
	3	1	3	
2	1	2	1	
	2	2	2	
	3	2	3	
3	1	3	1	
	2	3	2	
	3	3	3	

Step 4 Find $s = N(k)$ MOLS of order k. Denote these MOLS as $M_1, M_2, \ldots, M_s$. Note that $s < k$ for $k > 1$. Recall from Section 6.5 that there are at most $k - 1$ MOLS of order k and the maximum is achieved when k is prime.

Step 5 Fill Block 1 of column F_3 with entries from column 1 of M_1, Block 2 with entries from column 2 of M_1, and so on. If the number of blocks $b > k$, then reuse the columns of M_1 to fill rows in the remaining $(b - k)$ blocks. Repeat this procedure and fill columns F_4 through F_n using MOLS M_2 through M_s. If $s < n - 2$, then fill the remaining columns by randomly selecting the values of the corresponding factor. Using M_1 a sample table for $n = k = 3$ is shown below.

Block	Row	F_1	F_2	F_3
1	1	1	1	1
	2	1	2	2
	3	1	3	3
2	1	2	1	2
	2	2	2	3
	3	2	3	1
3	1	3	1	3
	2	3	2	1
	3	3	3	2

Step 6 The above table lists nine factor combinations, one corresponding to each row. Each combination can now be used to generate one or more tests as explained in Section 6.2.

```
End of Procedure PDMOLS
```

Configurations generated using the algorithm PDMOLS may need to be revised when there are constraints among factors.

In many practical situations, the table of configurations generated using the steps above, will need to be modified to handle constraints on factors and to account for factors that have fewer levels than k. There is no general algorithm for handling such special cases. Examples and exercises in the remainder of this chapter illustrate ways to handle a few situations.

For three factors, each having three levels, exhaustive pairwise testing would require 27 tests. However, the PDMOLS procedure can be used to generate only 9 tests that cover all pairwise interactions.

The PDMOLS procedure can be used to generate test configurations that ensure the coverage of all pairwise combinations of factor levels. It is easy to check that the number of test configurations so generated is usually much less than all possible combinations. For example, the total number of combinations is 27 for three factors each having three levels. However, as illustrated above, the number of test configurations derived using mutually orthogonal Latin squares is only nine.

Applications often impose constraints on factors such that certain level combinations are either not possible, or desirable, to achieve in practice. The next, rather long, example illustrates the use of PDMOLS in a more complex situation.

Example 6.17 DNA sequencing is a common activity amongst biologists and other researchers. Several genomics facilities are available that allow a DNA sample to be submitted for sequencing. One such facility is offered by the Applied Genomics Technology Center (AGTC) at the School of Medicine in Wayne State University. The submission of the sample itself is done using a software application available from AGTC. We refer to this software as AGTCS.

AGTCS is supposed to work on a variety of platforms that differ in their hardware and software configurations. Thus, the hardware platform and the operating system are two factors to be considered while developing a test plan for AGTCS. In addition, the user of AGTCS, referred to as Principal Investigator (PI), must either have a profile already created with AGTCS or create a new one prior to submitting a sample. AGTCS supports only a limited set of browsers. In all we have a total of four factors with their respective levels listed below.

Factor	Levels			
F1': Hardware (H)	PC	Mac		
F2': OS (O)	Windows 2000	Windows XP	OS 9	OS 10
F3': Browser (B)	Explorer	Nescape 4.x	Firefox	Mozilla
F4': PI (P)	New	Existing		

It is obvious that many more levels can be considered for each factor above. However, for the sake of simplicity, we constrain ourselves to the given list.

There are 64 combinations of the factors listed above. However, given the constraint that PCs and Macs will run their dedicated operating systems, the number of combinations reduces to 32, with 16 combinations each corresponding to the PC and the Mac. Note that each combination leads to a test configuration. However, instead of testing AGTCS for all 32 configurations, we are interested in testing under enough configurations so that all possible pairs of factor levels are covered.

We can now proceed to design test configurations in at least two ways. One way is to treat the testing on PC and Mac as two distinct problems and design the test configurations independently. Exercise 6.12 asks you to take this approach and explore its advantages over the second approach used in this example.

The approach used in this example is to arrive at a common set of test configurations that obey the constraint related to the operating systems. Let us now follow a modified version of procedure PDMOLS to generate the test configurations. We use $|F|$ to indicate the number of levels for factor F.

Input: $n = 4$ factors. $|F_1'| = 2, |F_2'| = 4, |F_3'| = 4,$ and $|F_4'| = 2$.
Output: A set of factor combinations such that all pairwise interactions are covered.

Step 1 Relabel the factors as F_1, F_2, F_3, and F_4 such that $|F_1| \geq |F_2| \geq |F_3| \geq |F_4|$. Doing so gives us $F_1 = F2'$, $F_2 = F3'$, $F_3 = F1'$, and $F_4 = F_4'$, $b = k = 4$. Note that a different assignment is also possible because $|F1| = |F_4|$ and $|F2| = |F_3|$. However, any assignment that satisfies the ordering constraint will retain the values of b and k as 4.

Step 2 Prepare a table containing 4 columns and $b \times k = 16$ rows divided into four blocks. Label the columns as $F_1, F_2, \ldots, F_n$. Each block contains k rows. The required table is shown below.

Block	Row	F_1(O)	F_2(B)	F_3(H)	F_4(P)
1	1				
	2				
	3				
	4				

Block	Row	F_1(O)	F_2(B)	F_3(H)	F_4(P)
2	1				
	2				
	3				
	4				
3	1				
	2				
	3				
	4				
4	1				
	2				
	3				
	4				

Step 3 Fill column F_1 with 1s in Block 1, 2s in Block 2, and so on. Fill Block 1 of column F_2 with the sequence $1, 2, \ldots, k$ in rows 1 through k. Repeat this for the remaining blocks. A sample table with columns F_1 and F_2 filled is shown below.

Block	Row	F_1(O)	F_2(B)	F_3(H)	F_4(P)
1	1	1	1		
	2	1	2		
	3	1	3		
	4	1	4		
2	1	2	1		
	2	2	2		
	3	2	3		
	4	2	4		
3	1	3	1		
	2	3	2		
	3	3	3		
	4	3	4		
4	1	4	1		
	2	4	2		
	3	4	3		
	4	4	4		

Step 4 Find MOLS of order 4. As 4 is not prime, we cannot use the procedure described in Example 6.13. In such a situation one may either resort to a predefined set of MOLS of

the desired order, or construct ones own using a procedure not described in this book but referred to in the Bibliographic Notes section. We follow the former approach and obtain the following set of three MOLS of order 4.

$$M_1 = \begin{matrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{matrix} \quad M_2 = \begin{matrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \\ 2 & 1 & 4 & 3 \end{matrix} \quad M_3 = \begin{matrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \end{matrix}$$

Step 5 Fill the remaining two columns of the table constructed in Step 3. As we have only two more columns to be filled, we use entries from M_1 and M_2. The final statistical design follows.

The test configurations constructed in Step 5 do not satisfy the constraints among the factors. Hence, the table needs to be modified.

Block	Row	F_1(O)	F_2(B)	F_3(H)	F_4(P)
1	1	1	1	1	1
	2	$\boxed{1}$	2	$\boxed{2}$	3*
	3	1	3	3*	4*
	4	1	4	4*	2
2	1	$\boxed{2}$	1	$\boxed{2}$	2
	2	2	2	1	4*
	3	2	3	4*	3*
	4	2	4	3*	1
3	1	3	1	3*	3*
	2	3	2	4*	1
	3	$\boxed{3}$	3	$\boxed{1}$	2
	4	3	4	2	4*
4	1	4	1	4*	4*
	2	4	2	3*	2
	3	4	3	2	1
	4	$\boxed{4}$	4	$\boxed{1}$	3*

A boxed entry indicates a pair that does not satisfy the operating system constraint. An entry marked with an asterisk (*) indicates an invalid level.

Step 6 Using the 16 entries in the table above, we can obtain 16 distinct test configurations for AGTCS. However, we need to resolve two problems before we get to the design of test configurations. One problem is that factors F_3 and F_4 can only assume values 1 and 2, whereas the table above contains other infeasible values for these two factors. These infeasible values are marked with an asterisk (*).

One simple way to get rid of the infeasible values is to replace them by an arbitrarily selected feasible value for the corresponding factor.

The other problem is that some configurations do not satisfy the operating system constraint. Four such configurations are highlighted in the design above by enclosing the corresponding numbers in rectangles. For example, the entry in Block 3, Row 3 indicates that factor F_3, that corresponds to `Hardware`, is set at level `PC` while factor F_1, that corresponds to `Operating System`, is set to `Mac OS 9`.

Obviously, we cannot delete these rows as that would leave some pairs uncovered. For example, removing Block 3, Row 3 will leave the following five pairs uncovered: $(F_1 = 3, F_2 = 3)$, $(F_1 = 3, F_4 = 2)$, $(F_2 = 3, F_3 = 1)$, $(F_2 = 3, F_4 = 2)$, and $(F_3 = 1, F_4 = 2)$.

Modification of entries in a row might render some pair uncovered. When this happens additional rows will need to be added to the table to ensure complete coverage.

We follow a two step procedure to remove the highlighted configurations and retain complete pairwise coverage. In the first step, we modify the four highlighted rows so they do not violate the constraint. However, by doing so we leave some pairs uncovered. In the second step, we add new configurations that cover the pairs that are left uncovered when we replace the highlighted rows.

The modified design is as follows. In this design, we have replaced the infeasible entries for factors F_3 and F_4 by feasible ones (also see Exercise 6.10). Entries changed to resolve conflicts are highlighted. Also, "don't care" entries are marked with a dash (–). While designing a configuration, a don't care entry can be selected to create any valid configuration.

It is easy to obtain 20 test configurations from the design given above. Recall that we would get a total of 32 configurations if a brute force method is used. However, by using the PDMOLS procedure, we have been able to achieve a reduction of 12 test configurations. A further reduction can be obtained in the number of test configurations by removing row 2 in block 5 as it is redundant in the presence of row 1 in block 4 (also see Exercise 6.11).

Block	Row	F_1(O)	F_2(B)	F_3(H)	F_4(P)
1	1	1	1	1	1
	2	1	2	$\boxed{1}$	1
	3	1	3	1	2
	4	1	4	2	2
2	1	2	1	$\boxed{1}$	2
	2	2	2	1	1
	3	2	3	1	2
	4	2	4	2	1
3	1	3	1	1	1
	2	3	2	2	1
	3	$\boxed{1}$	3	1	2
	4	3	4	2	2
4	1	4	1	2	2
	2	4	2	1	2
	3	4	3	2	1
	4	$\boxed{2}$	4	1	1
5	1	–	2	2	1
	2	–	1	2	2
	3	3	3	–	2
	4	4	4	–	1

Entries changed to satisfy the operating system constraint are boxed. Don't care entries are marked as – and can be set to an appropriate level for the corresponding factor. Care must be taken not to violate the operating system constraint while selecting a value for a don't care entry.

6.7.1 Shortcomings of using MOLS for test design

While the approach of using mutually orthogonal Latin squares to develop test configurations has been used in practice, it has its shortcomings as enumerated below.

While MOLS can be used in the design of combinatorial objects, the process has disadvantages. First, a sufficient number of MOLS of the desired order might not exist, and second the test set so generated is generally larger than what can be obtained using some other methods described later.

1. *A sufficient number of MOLS might not exist for the problem at hand.* As an illustration, in Example 6.17, we needed only two MOLS of order 4 which do exist. However, if the number of factors increases to, say 6, without any change in the value of $k = 4$, then we would be short by one MOLS as there exist only three MOLS of order 4. As mentioned earlier in Step 5 in Procedure PDMOLS, the lack of sufficient MOLS is compensated by randomizing the generation of columns for the remaining factors.
2. *While the MOLS approach assists with the generation of a balanced design in that all interaction pairs are covered an equal number of times, the number of test configurations is often larger than what can be achieved using other methods.*

For example, the application of Procedure PDMOLS to the problem in Example 6.16 leads to a total of nine test cases for testing the GUI. This is in contrast to the six test cases generated by the application of Procedure SAMNA.

Several other methods for generating combinatorial designs have been proposed for use in software testing. The most commonly used are based on the notions of orthogonal arrays, covering arrays, mixed-level covering arrays, and in-parameter order. These methods are described in the following sections.

6.8 Orthogonal Arrays

An orthogonal array of strength t is an $N \times k$ array where any $N \times t$ subarray contains a t-tuple exactly the same number of times. All factors in an orthogonal array have exactly the same number of levels.

An orthogonal array is an $N \times k$ matrix in which the entries are from a finite set S of s symbols such that any $N \times t$ subarray contains each t-tuple exactly the same number of times. Such an orthogonal array is denoted by $OA(N, k, s, t)$. The index of an orthogonal array, denoted by λ, is equal to N/s^t. N is referred to as the number of *runs* and t as the *strength* of the orthogonal array.

When using orthogonal arrays in software testing, each column corresponds to a factor and the elements of a column to the levels for the corresponding factor. Each run, or row of the orthogonal array, leads to a test case or a test configuration. The following example illustrates some properties of simple orthogonal arrays.

Each row in an orthogonal array is also know as a run.

Example 6.18 The following is an orthogonal array having 4 runs and has a strength of 2. It uses symbols from the set {1, 2}. Notationally, this array is denoted as $OA(4, 3, 2, 2)$. Note that the value of parameter k is 3 and hence we have labeled the columns as F_1, F_2, and F_3 to indicate the three factors.

Run	F_1	F_2	F_3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

The index λ of the array shown above is $4/2^2 = 1$ implying that each pair ($t = 2$) appears exactly once ($\lambda = 1$) in any 4×2 subarray. There is a total of $s^t = 2^2 = 4$ pairs given as (1, 1), (1, 2), (2, 1), and (2, 2). It is easy to verify, by a mere scanning of the array, that each of the four pairs appears exactly once in each 4×2 subarray.

The orthogonal array shown below has 9 runs and a strength of 2. Each of the four factors can be at any one of three levels. This array is denoted as $OA(9, 4, 3, 2)$ and has an index of 1.

Run	F_1	F_2	F_3	F_4
1	1	1	1	1
2	1	2	2	3
3	1	3	3	2
4	2	1	2	2
5	2	2	3	1
6	2	3	1	3
7	3	1	3	3
8	3	2	1	2
9	3	3	2	1

There are nine possible pairs of symbols formed from the set {1, 2, 3}. Again, it is easy to verify that each of these pairs occurs exactly once in any 9×2 subarray taken out of the matrix given above.

An alternate notation for orthogonal arrays is also used. For example, $L_N(s^k)$ denotes an orthogonal array of N runs where k factors take on any value from a set of s symbols. Here, the strength t is assumed to be 2. Using this notation, the two arrays in Example 6.18 are denoted by $L_4(2^3)$ and $L_9(3^3)$, respectively.

Sometimes a simpler notation L_N is used that ignores the remaining three parameters k, s, and t assuming that they can be determined from the context. Using this notation, the two arrays in Example 6.18 are denoted by L_4 and L_9, respectively. We prefer the notation $OA(N, k, s, t)$.

6.8.1 Mixed-level orthogonal arrays

The orthogonal arrays shown in Example 6.18 are also known as *fixed-level orthogonal arrays*. This is because the design of such arrays assumes that all factors assume values from the same set of s values. As illustrated in Section 6.1.2, this is not true in many practical situations. In many practical applications, one encounters more than one factor, each taking on a different set of values. Mixed-level orthogonal arrays are useful in designing test configurations for such applications.

A mixed level orthogonal array contains N runs of k factors where each factor may have different number of levels.

A mixed-level orthogonal array of strength t is denoted by $MA(N, s_1^{k_1} s_2^{k_2}, \ldots, s_p^{k_p}, t)$ indicating N runs where k_1 factors are at s_1 levels, k_2 factors at s_2 levels, and so on. The total number of factors is $\Sigma_{i=1}^{p} k_i$.

Mixed-level orthogonal arrays retain the balance property, i.e., each $N \times t$ subarray contains each t-tuple exactly the same number of times.

The formula used for computing the index λ of an orthogonal array does not apply to the mixed-level orthogonal array as the count of values for each factor is a variable. However, the balance property of orthogonal arrays remains intact for mixed-level orthogonal arrays in that any $N \times t$ subarray contains each t-tuple corresponding to the t columns, exactly the same number of times, which is λ. The next example shows two mixed-level orthogonal arrays.

Example 6.19 A mixed-level orthogonal array $MA(8, 2^4 4^1, 2)$ is shown below. It can be used to design test configurations for an application that contains four factors each at two levels and one factor at four levels.

Run	F_1	F_2	F_3	F_4	F_5
1	1	1	1	1	1
2	2	2	2	2	1
3	1	1	2	2	2
4	2	2	1	1	2
5	1	2	1	2	3
6	2	1	2	1	3
7	1	2	2	1	4
8	2	1	1	2	4

Notice that the above array is balanced in the sense that in any subarray of size 8×2, each possible pair occurs exactly the same number of times. For example, in the two leftmost columns, each pair occurs exactly twice. In columns 1 and 3, each pair also occurs exactly twice. In columns 1 and 5, each pair occurs exactly once.

Mixed-level orthogonal arrays can be used to generate tests when all factors may not have the same number of levels.

The next example shows $MA(16, 2^6 4^3, 2)$. This array can be used to generate test configurations when there are six binary factors, labeled F_1 through F_6 and three factors each with four possible levels, labeled F_7 through F_9.

Run	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9
1	1	1	1	1	1	1	1	1	1
2	2	2	1	2	1	2	1	3	3
3	1	2	2	2	2	1	3	1	3
4	2	1	2	1	2	2	3	3	1
5	1	1	2	2	2	2	1	4	4
6	2	2	2	1	2	1	1	2	2
7	1	2	1	1	1	2	3	4	2
8	2	1	1	2	1	1	3	2	4

Run	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9
9	2	2	1	1	2	2	4	1	4
10	1	1	1	2	2	1	4	3	2
11	2	1	2	2	1	2	2	1	2
12	1	2	2	1	1	1	2	3	4
13	2	2	2	2	1	1	4	4	1
14	1	1	2	1	1	2	4	2	3
15	2	1	1	1	2	1	2	4	3
16	1	2	1	2	2	2	2	2	1

Example 6.20 In Example 6.2, we have three factors at two levels and one factor at three levels. The following mixed array, $MA(12, 2^3, 3^1, 2)$ can be used to generate 12 test configurations. This implies that a total of 12 configurations are required for exhaustive testing of the pizza service.

Run	Size	Toppings	Address	Phone
1	1	1	1	1
2	1	1	2	1
3	1	2	1	2
4	1	2	2	2
5	2	1	1	2
6	2	1	2	2
7	2	2	1	1
8	2	2	2	1
9	3	1	1	2
10	3	1	2	1
11	3	2	1	1
12	3	2	2	2

Following is the set of test inputs for the pizza service derived using $MA(12, 2^3, 3^1, 2)$ shown above. To arrive at the design below, we assumed that levels 1, 2, and 3 for **Size** correspond to `large`, `medium`, and `Small`, respectively. Similarly, levels 1 and 2, corresponding to the remaining three factors, are mapped to their respective values in Example 6.2 with a 1 corresponding to the leftmost value and a 2 to the rightmost value. Note that this assignment of integers to actual values is arbitrary and does not affect in any way the generated set of test configurations.

Run	Size	Toppings	Address	Phone
1	Large	Custom	Valid	Valid
2	Large	Custom	Invalid	Valid
3	Large	Preset	Valid	Invalid
4	Large	Preset	Invalid	Invalid
5	Medium	Custom	Valid	Invalid
6	Medium	Custom	Invalid	Invalid
7	Medium	Preset	Valid	Valid
8	Medium	Preset	Invalid	Valid
9	Small	Custom	Valid	Invalid
10	Small	Custom	Invalid	Valid
11	Small	Preset	Valid	Valid
12	Small	Preset	Invalid	Invalid

It is easy to check that all possible pairs of factor combinations are covered in the design above. Testing the pizza service under the 12 configurations above will likely reveal any pairwise interaction errors.

While the balance requirement is essential in many statistical experiments, it is not likely to be a requirement in software testing. This observation leads to other techniques for generating tests using combinatorial designs.

So far we have examined techniques for generating balanced combinatorial designs and explained how to use these to generate test configurations and test sets. However, while the balance requirement is often essential in statistical experiments, it is not always so in software testing. For example, if a software application has been tested once for a given pair of factor levels, there is generally no need for testing it again for the same pair, unless the application is known to behave non-deterministically. One might also test the application against the same pair of values to ensure repeatability of results. Thus, for deterministic applications, and when repeatability is not the focus of the test, we can relax the balance requirement and instead use *covering arrays*, or *mixed-level covering arrays* for obtaining combinatorial designs. This is the topic of the next section.

A covering array is an $N \times k$ matrix in which each $N \times t$ subarray contains a t-tuple at least a given number of times. In software testing we assume that each $N \times t$ aubaray contains each possible t-tuple at least once.

6.9 Covering and Mixed-Level Covering Arrays

A covering array $CA(N, k, s, t)$ is an $N \times k$ matrix in which entries are from a finite set S of s symbols such that each $N \times t$ subarray contains each possible t-tuple *at least* λ times. As in the case of orthogonal arrays, N denotes the number of runs, k the number factors, s the number of levels for each factor, t the strength, and λ the index. While generating test cases or test configurations for a software application, we use $\lambda = 1$.

Let us point to a key difference between a covering array and an orthogonal array. While an orthogonal array $OA(N, k, s, t)$ covers each possible t-tuple λ times in any $N \times t$ subarray, a covering array $CA(N, k, s, t)$ covers each possible t-tuple at least λ times in any $N \times t$ subarray. Thus covering arrays do not meet the balance requirement that is met by orthogonal arrays. This difference leads to combinatorial designs that are often smaller in size than orthogonal arrays. Covering arrays are also referred to as *unbalanced* designs.

Covering arrays do not meet the balance requirement as different t-tuples in an $N \times t$ subarray might occur different number of times. Thus, covering arrays are also known as unbalanced designs.

Of course, we are interested in *minimal* covering arrays. These are covering arrays that satisfy the requirement of a covering array in the least number of runs.

Example 6.21 A balanced design of strength 2 for 5 binary factors, requires 8 runs and is denoted by $OA(8, 5, 2, 2)$. However, a covering design with the same parameters requires only 6 runs. Thus, in this case, we save two test configurations if we use a covering design instead of the orthogonal array. The two designs follow.

Designs based on covering arrays are generally smaller than those based on orthogonal arrays.

$OA(8, 5, 2, 2)$=

Run	F_1	F_2	F_3	F_4	F_5
1	1	1	1	1	1
2	2	1	1	2	2
3	1	2	1	2	1
4	1	1	2	1	2
5	2	2	1	1	2
6	2	1	2	2	1
7	1	2	2	2	2
8	2	2	2	1	1

$CA(6, 5, 2, 2)$=

Run	F_1	F_2	F_3	F_4	F_5
1	1	1	1	1	1
2	2	2	1	2	1
3	1	2	2	1	2
4	2	1	2	2	2
5	2	2	1	1	2
6	1	1	1	2	2

6.9.1 Mixed-level covering arrays

Mixed-level covering arrays are analogous to mixed-level orthogonal arrays in that both are practical designs that allow the use of factors to assume levels from different sets. A mixed-level covering array is denoted as $MCA(N, s_1^{k_1} s_2^{k_2}, \ldots, s_p^{k_p}, t)$ and refers to an $N \times Q$ matrix of entries such that, $Q = \Sigma_{i=1}^{p} k_i$ and each $N \times t$ subarray contains at least one occurrence of each t-tuple corresponding to the t columns.

A mixed-level covering array has factors where each may assume different number of values. Such arrays are generally smaller than mixed-level orthogonal arrays.

Mixed-level covering arrays are generally smaller than mixed-level orthogonal arrays and more appropriate for use in software testing. The next example, shows $MCA(6, 2^3 3^1, 2)$. Comparing this with $MA(12, 2^3 3^1, 2)$ from Example 6.20, we notice a reduction of 6 test inputs.

Run	Size	Toppings	Address	Phone
1	1	1	1	1
2	2	2	1	2
3	3	1	2	2
4	1	2	2	2
5	2	1	2	1
6	3	2	1	1

Notice that the array above is not balanced and hence is not a mixed-level orthogonal array. For example, the pair (1, 1) occurs twice in columns *Address* and *Phone* but the pair (1, 2) occurs only once in the same two columns. However, this imbalance is unlikely to effect the reliability of the software test process as each of the four possible pairs between *Address* and *Phone* are covered and hence will occur in a test input. For the sake of completeness, we list below the set of six tests for the pizza service.

Run	Size	Toppings	Address	Phone
1	Large	Custom	Valid	Valid
2	Medium	Preset	Valid	Invalid
3	Small	Custom	Invalid	Invalid
4	Large	Preset	Invalid	Invalid
5	Medium	Custom	Invalid	Valid
6	Small	Preset	Valid	Valid

6.10 Arrays of Strength > 2

In the previous sections, we have examined various combinatorial designs all of which have a strength $t = 2$. While tests generated from such designs are targeted at discovering errors due to pairwise interactions, designs with higher strengths are sometimes needed to achieve higher confidence in the correctness of software. Thus, for example, a design with $t = 3$ will lead to tests that target errors due to 3-way interactions. Such designs are usually more expensive than designs with $t = 2$ in terms of the number of tests they generate. The next example illustrates a design with $t = 3$.

Example 6.22 Pacemakers are medical devices used for automatically correcting abnormal heart conditions. These devices are controlled by software that encodes several complicated algorithms for

sensing, computing, and controlling the heart rate. Testing of a typical modern pacemaker is an intense and complex activity. Testing it against a combination of input parameters is one aspect of this activity. The table below lists only five of the several parameters, and their values, that control the operation of a pacemaker.

Parameter	**Levels**		
Pacing mode	AAI	VVI	DDD-R
QT interval	Normal	Prolonged	Shortened
Respiratory rate	Normal	Low	High
Blood pressure	Normal	Low	High
Body temperature	Normal	Low	High

Due to the high reliability requirement of the pacemaker, we would like to test it to ensure that there are no pairwise or 3-way interaction errors. Thus, we need a suitable combinatorial object with strength 3. We could use an orthogonal array $OA(54, 5, 3, 3)$ that has 54 runs for five factors each at three levels and is of strength 3. Thus, a total of 54 tests will be required to test for all 3-way interactions of the five pacemaker parameters (see also Exercises 6.16 and 6.17).

It is worth noting that a combinatorial array for three or more factors that provides 2-way coverage, balanced or unbalanced, also provides some n-way coverage, for each $n > 2$. For example, $MA(12, 2^3 3^1, 2)$ covers all 20 of the 32 3-way interactions of the three binary and one ternary parameters. However, the mixed-level covering array $MCA(6, 2^3 3^1, 2)$ covers only 12 of the 20 3-way interactions. While (**Size**=Large, **Toppings**= Custom, **Address**=Invalid) is a 3-way interaction covered by MCA $(12, 2^3 3^1, 2)$, and the triple (**Size**=Medium, **Toppings**=Custom, **Address**=Invalid) is not covered.

A combinatorial array for three or more factors that provides pairwise coverage may also cover some higher order interactions.

6.11 Generating Covering Arrays

Several practical procedures have been developed and implemented for the generation of various types of designs considered in this chapter. Here, we focus on a procedure for generating mixed-level covering arrays for pairwise deigns.

The procedure described below, is known as *In-Parameter-Order*, or simply IPO, procedure. It takes the number of factors and the corresponding

levels as inputs and generates a covering array that is, at best, near optimal. The covering array generated covers all pairs of input parameters at least once and is used for generating tests for pairwise testing. The term "parameter" in "IPO" is a synonym for "factor."

The In-order-parameter (IPO) procedure is a practical means for constructing mixed-level covering arrays.

The IPO procedure has several variants only one of which is presented here. The entire procedure is divided into three parts: the main procedure, named IPO, procedure HG for horizontal growth, and procedure VG for vertical growth. While describing IPO and its sub procedures, we use *parameter* for *factor* and *value* for *level*.

`Procedure for generating pairwise mixed-level covering designs.`

Input: (a) $n \geq 2$: Number of input parameters. (b) Number of values for each parameter.

Output: CA: A set of parameter combinations such that all pairs of values are covered at least once.

Procedure: `IPO`

/*
- $F_1, F_2, \ldots, F_n$ denote n parameters. $q_1, q_2, \ldots, q_n$ denote the number of values for the corresponding parameter.
- $\mathcal{T}$ holds partial or complete runs of the form $(v_1, v_2, \ldots, v_k), 1 \leq k \leq n$, where v_i denotes a value for parameter $F_i, 1 \leq i \leq k$.
- $\mathcal{D}(F)$ denotes the domain of parameter F, i.e. the set of all values for parameter F.

*/

The IPO procedure consists of two phases. In the first phase the desire array grows horizontally by the addition of levels to each parameter. In phase two, additional rows are added to the array to include pairs uncovered thus far.

Step 1 [*Initialize*] Generate all pairs for parameters F_1 and F_2. Set $\mathcal{T} = \{(r, s) | \text{for all } r \in \mathcal{D}(F_1) \text{ and } s \in \mathcal{D}(F_2)\}$.

Step 2 [*Possible termination*] `if` $n = 2$ `then` set $CA = \mathcal{T}$ and terminate the algorithm, else continue.

Step 3 [*Add remaining parameters*] Repeat the following steps for parameters $F_k, k = 3, 4, \ldots, n$.

3.1 [*Horizontal growth*] Replace each partial run $(v_1, v_2, v_{k-1}) \in \mathcal{T}$ with (v_1, v_2, v_{k-1}, v_k), where v_k is a suitably selected value of parameter F_k. More details of this step are found in procedure `HG` described later in this section.

3.2 [*Uncovered pairs*] Compute the set U of all uncovered pairs formed by pairing parameters $F_i, 1 \leq i \leq k-1$ and parameter F_k.

3.3 [*Vertical growth*] If U is empty, then terminate this step, else continue. For each uncovered pair $u = (v_j, v_k) \in U$, add a run $(v_1, v_2, \ldots, v_j, \ldots, v_{k-1}, v_k)$ to $\mathcal{T}$. Here v_j and v_k denote the values

of parameters F_j and F_k, respectively. More details of this step are found in procedure VG described later in this section.

```
End of Procedure IPO
```

```
Procedure for horizontal growth.
```

Input: (a) $\mathcal{T}$: A set of $m \geq 1$ runs of the kind $R = (v_1, v_2, \ldots, v_{k-1}), k > 2$ where $v_i, 1 \leq i \leq (k-1)$ is a value of parameter F_i. (b) Parameter $F \neq F_i, 1 \leq i \leq (k-1)$.

Output: $\mathcal{T}'$: A set of runs $(v_1, v_2, \ldots, v_{k-1}, v_k), k > 2$ obtained by extending the runs in $\mathcal{T}$ that cover the maximum number of pairs between parameter $F_i, 1 \leq i \leq (k-1)$ and parameter F.

Procedure: HG

/*
- $\mathcal{D}(F) = \{l_1, l_2, \ldots, l_q\}, q \geq 1$.
- $t_1, t_2, \ldots, t_m$ denote the $m \geq 1$ runs in $\mathcal{T}$.
- For a run $t \in \mathcal{T}$, where $t = (v_1, v_2, \ldots, v_{k-1})$, $extend(t, v) = (v_1, v_2, \ldots, v_{k-1}, v)$, where v is a value of parameter F.
- Given $t = (v_1, v_2, \ldots, v_{k-1})$ and v is a parameter value, $pairs(extend(t, v)) = \{(v_i, v), 1 \leq i \leq (k-1)\}$.

*/

Step 1 Let $AP = \{(r, s)|$, where r is a value of parameter $F_i, 1 \leq i \leq (k-1)$ and $s \in \mathcal{D}(F)\}$. Thus AP is the set of all pairs formed by combining parameters $F_i, 1 \leq i \leq (k-1)$, taken one at a time, with parameter F.

Step 2 Let $\mathcal{T}' = \emptyset$. $\mathcal{T}'$ denotes the set of runs obtained by extending the runs in $\mathcal{T}$ in the following steps.

Step 3 Let $C = min(q, m)$. Here C is the number of elements in the set $\mathcal{T}$ or the set $\mathcal{D}(F)$, whichever is less.

Step 4 Repeat the next two sub steps for $j = 1$ to C.

4.1 Let $t'_j = extend(t_j, l_j)$. $\mathcal{T}' = \mathcal{T}' \cup t'_j$.

4.2 $AP = AP - pairs(t'_j)$.

Step 5 If $C = m$, then return $\mathcal{T}'$.

Step 6 We will now extend the remaining runs in $\mathcal{T}$ by values of parameter F that cover the maximum pairs in AP. Repeat the next four sub steps for $j = C + 1$ to m.

6.1 Let $AP' = \emptyset$ and $v' = l_1$.

6.2 In this step, we find a value of F by which to extend run t_j. The value that adds the maximum pairwise coverage is selected. Repeat the next two sub steps for each $v \in \mathcal{D}(F)$.

6.2.1 $AP'' = \{(r, v)|,$ where r is a value in run $t_j\}$. Here AP'' is the set of all new pairs formed when run t_j is extended by v.

6.2.2 If $|AP''| > |AP'|$, then $AP' = AP''$ and $v' = v$.

6.3 Let $t'_j = extend(t_j, v')$. $\mathcal{T}' = \mathcal{T}' \cup t'_j$.

6.4 $AP = AP - AP'$.

Step 7 Return $\mathcal{T}'$.

`End of Procedure HG`

`Procedure for vertical growth.`

Input: (a) $\mathcal{T}$: A set of $m \geq 1$ runs each of the kind $(v_1, v_2, \ldots, v_{k-1}, v_k), k > 2$, where $v_i, 1 \leq i \leq k$, is a value of parameter F_i. (b) The set MP of all pairs (r, s), where r is a value of parameter $F_i, 1 \leq i \leq (k-1)$, $s \in \mathcal{D}(F_k)$, and the pair (r, s) is not contained in any run in $\mathcal{T}$.

Output: A set of runs $(v_1, v_2, \ldots, v_{k-1}, v_k), k > 2$ such that all pairs obtained by combining values of parameter $F_i, 1 \leq i \leq (k-1)$ with parameter F_k are covered.

Procedure: `VG`

/*
- $\mathcal{D}(F) = \{l_1, l_2, \ldots, l_q\}, q \geq 1$.
- $t_1, t_2, \ldots, t_m$ denote the $m \geq 1$ runs in $\mathcal{T}$.
- $(A_i.r, B_j.s)$ denotes a pair of values r and s that correspond to parameters A and B, respectively.
- In run $(v_1, v_2, \ldots, v_{i-1}, *, v_{i+1}, \ldots, v_k), i < k$, a "*" denotes a don't care value for parameter F_i. When needed, we use dc instead of "*".

*/

Step 1 Let $\mathcal{T}' = \emptyset$.

Step 2 Add new tests to cover the uncovered pairs.
For each missing pair $(F_i.r, F_k.s) \in MP, 1 \leq i < k$, repeat the next two sub steps.

2.1 If there exists a run $(v_1, v_2 \ldots, v_{i-1}, *, v_{i+1}, \ldots, v_{k-1}, s) \in \mathcal{T}'$, then replace it by the run $(v_1, v_2, \ldots, v_{i-1}, r, v_{i+1}, \ldots, s)$ and examine the next missing pair, else go to the next sub step.

2.2 Let $t = (dc_1, dc_2, \ldots, dc_{i-1}, r, dc_{i+1}, \ldots, dc_{k-1}, s), 1 \leq i < k$. Add t to $\mathcal{T}'$.

Step 3 For each run $t \in \mathcal{T}'$, replace any don't care entry by an arbitrarily selected value of the corresponding parameter. Instead, one could also select a value that maximizes the number of higher order tuples such as triples.

Step 4 Return $\mathcal{T} \cup \mathcal{T}'$.

```
End of Procedure VG
```

Example 6.23 Suppose it is desired to find a mixed covering design $MCA(N, 2^1 3^2, 2)$. Let us name the three parameters as A, B, and C, where parameters A and C each have three values and parameter B has two values. The domains of A, B, and C are, respectively, $\{a_1, a_2, a_3\}$, $\{b_1, b_2\}$, and $\{c_1, c_2, c_3\}$.

IPO Step 1: Note that $n = 3$. In this step construct all runs that consist of pairs of values of the first two parameters, A and B. This leads to the following set:

$\mathcal{T} = \{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2), (a_3, b_1), (a_3, b_2)\}$. We denote the elements of this set, starting from the first element, as $t_1, t_2, \ldots, t_6$.

As $n \neq 2$, continue to IPO.Step 3. Execute the loop for $k = 3$. In the first sub step inside the loop, perform horizontal growth (Step 3.1) using procedure HG. At HG.Step 1, compute the set of all pairs between parameters A and C, and parameters B and C. This leads us to the following set of 15 pairs.

$$\begin{aligned} AP = \{&(a_1, c_1), (a_1, c_2), (a_1, c_3), (a_2, c_1), (a_2, c_2), (a_2, c_3), (a_3, c_1), (a_3, c_2), \\ &(a_3, c_3), (b_1, c_1), (b_1, c_2), (b_1, c_3), (b_2, c_1), (b_2, c_2), (b_2, c_3)\} \end{aligned}$$

Next, in HG.Step 2 initialize $\mathcal{T}'$ to $\emptyset$. At HG.Step 3 compute $C = min(q, m) = min(3, 6) = 3$. (Do not confuse this C with the parameter C.) Now move to HG.Step 4 where the extension of the runs in $\mathcal{T}$ begins ith $j = 1$.

HG.Step 4.1: $t'_1 = extend(t_1, l_1) = (a_1, b_1, c_1)$. $\mathcal{T}' = \{(a_1, b_1, c_1)\}$.

HG.Step 4.2: The run just created by extending t_1 has covered the pairs (a_1, c_1) and (b_1, c_1). Now update AP as follows.

$$\begin{aligned} AP &= AP - \{(a_1, c_1), (b_1, c_1)\} \\ &= \{(a_1, c_2), (a_1, c_3), (a_2, c_1), (a_2, c_2), (a_2, c_3), (a_3, c_1), \\ &\quad (a_3, c_2), (a_3, c_3), (b_1, c_2), (b_1, c_3), (b_2, c_1), (b_2, c_2), (b_2, c_3)\} \end{aligned}$$

Repeating the above sub steps for $j = 2$ and $j = 3$, we obtain the following.

HG.Step 4.1: $t'_2 = extend(t_2, l_2) = (a_1, b_2, c_2)$. $\mathcal{T}' = \{(a_1, b_1, c_1), (a_1, b_2, c_2)\}$.

HG.Step 4.2:

$$\begin{aligned} AP &= AP - \{(a_1, c_2), (b_2, c_2)\} \\ &= \{(a_1, c_3), (a_2, c_1), (a_2, c_2), (a_2, c_3), (a_3, c_1), (a_3, c_2), (a_3, c_3) \\ &\quad (b_1, c_2), (b_1, c_3), (b_2, c_1), (b_2, c_3)\} \end{aligned}$$

HG.Step 4.1: $t'_3 = extend(t_3, l_3) = (a_2, b_1, c_3)$. $\mathcal{T}' = \{(a_1, b_1, c_1), (a_1, b_2, c_2), (a_2, b_1, c_3)\}$.

HG.Step 4.2:

$$\begin{aligned} AP &= AP - \{(a_2, c_3), (b_1, c_3)\} \\ &= \{(a_1, c_3), (a_2, c_1), (a_2, c_2), (a_3, c_1), (a_3, c_2), (a_3, c_3). \\ &\quad (b_1, c_2), (b_2, c_1), (b_2, c_3)\} \end{aligned}$$

At HG.Step 5 $C \neq 6$ and hence continue at HG.Step 6. Repeat the sub steps for $j = 4, 5$, and 6. Let us first do it for $j = 4$.

HG.Step 6.1: $AP' = \emptyset$ and $v' = c_1$. Next, move to HG.Step 6.2. In this step find the best value of parameter C for extending run t_j. This step is to be repeated for each value of parameter C. Let us begin with $v = c_1$.

HG.Step 6.2.1: If run t_4 by v were to be extended, we get t (a_2, b_2, c_1). This run covers two pairs in AP, namely (a_2, c_1) and (b_2, c_1). Hence, $AP'' = \{(a_2, c_1), (b_2, c_1)\}$.

HG.Step 6.2.2: As $|AP''| > |AP'|$, set $AP' = \{(a_2, c_1), (b_2, c_1)\}$ and $v' = c_1$. Next repeat the two sub steps 6.2.1 and 6.2.2 for $v = c_2$ and then $v = c_3$.

HG.Step 6.2.1: Extending t_4 by $v = c_2$, we get the run (a_2, b_2, c_2). This extended run leads to one pair in AP. Hence $AP'' = \{(a_2, c_2)\}$.

HG.Step 6.2.2: As $|AP''| < |AP'|$, do not change AP'.

HG.Step 6.2.1: Now extending t_4 by $v = c_3$, we get the run (a_2, b_2, c_3). This extended run gives us (b_2, c_3) in AP. Hence, $AP'' = \{(b_2, c_3)\}$.

HG.Step 6.2.2: As $|AP''| < |AP'|$, do not change AP'. This completes the execution of the inner loop. We have found that the best way to extend t_4 is to do so by c_1.

HG.Step 6.3: $t'_4 = extend(t_4, c_1) = (a_2, b_2, c_1)$.

$$\mathcal{T}' = \{(a_1, b_1, c_1), (a_1, b_2, c_2), (a_2, b_1, c_3), (a_2, b_2, c_1)\}.$$

HG.Step 6.4: Update AP by removing the pairs covered by t'_4.

$$\begin{aligned} AP &= AP - \{(a_2, c_1), (b_2, c_1)\} \\ &= \{(a_1, c_3), (a_2, c_2), (a_3, c_1), (a_3, c_2), (a_3, c_3) \\ &\quad (b_1, c_2), (b_2, c_3)\} \end{aligned}$$

Let us move to the case $j = 5$ and find the best value of parameter C to extend t_5.

HG.Step 6.1: $AP' = \emptyset$, $v' = c_1$, and $v = c_1$.

HG.Step 6.2.1: If run t_5 by v were to be extended, we get the run (a_3, b_1, c_1). This run covers one pair in the updated AP, namely (a_3, c_1). Hence, $AP'' = \{(a_3, c_1)\}$.

HG.Step 6.2.2: As $|AP''| > |AP'|$, set $AP' = \{(a_3, c_1)\}$ and $v' = c_1$.

HG.Step 6.2.1: Extending t_5 by $v = c_2$ leads to the run (a_3, b_1, c_2). This extended run gives us two pairs, (a_3, c_2) and (b_1, c_2) in the updated AP. Hence $AP'' = \{(a_3, c_2), (b_1, c_2)\}$.

HG.Step 6.2.2: As $|AP''| > |AP'|$, $AP' = \{(a_3, c_2), (b_1, c_2)\}$ and $v' = c_2$.

HG.Step 6.2.1: Next, extend t_5 by $v = c_3$, and get (a_3, b_1, c_3). This extended run gives the pair (a_3, c_3) in AP. Hence, $AP'' = \{(a_3, c_3)\}$.

HG.Step 6.2.2: As $|AP''| < |AP'|$, do not change $AP' = \{(a_3, c_2), (b_1, c_2)\}$. This completes the execution of the inner loop. We have found that the best way to extend t_5 is to do so by c_2. Notice that this is not the only best way to extend t_5.

HG.Step 6.3: $t'_5 = extend(t_5, c_2) = (a_3, b_1, c_2)$.

$$\mathcal{T}' = \{(a_1, b_1, c_1), (a_1, b_2, c_2), (a_2, b_1, c_3), (a_2, b_2, c_1), (a_3, b_1, c_2\}.$$

HG.Step 6.4: Update AP by removing the pairs covered by t'_5.

$$\begin{aligned} AP &= AP - \{(a_3, c_2), (b_1, c_2)\} \\ &= \{(a_1, c_3), (a_2, c_2), (a_3, c_1), (a_3, c_3), (b_2, c_3)\} \end{aligned}$$

Finally, we move to the last iteration of the outer loop in HG, execute it for $j = 6$, and find the best value of parameter C to extend t_6.

HG.Step 6.1: $AP' = \emptyset$, $v' = c_1$, and $v = c_1$.

HG.Step 6.2.1: If run t_6 by v were to be extended, we get the run (a_3, b_2, c_1). This run covers one pair in the updated AP, namely (a_3, c_1). Hence, $AP'' = \{(a_3, c_1)\}$.

HG.Step 6.2.2: As $|AP''| > |AP'|$, set $AP' = \{(a_3, c_1)\}$ and $v' = c_1$.

HG.Step 6.2.1: Extending t_6 by $v = c_2$, we get the run (a_3, b_2, c_2). This extended run gives us no pair in the updated AP. Hence $AP'' = \emptyset$.

HG.Step 6.2.2: As $|AP''| < |AP'|$, AP' and v' remain unchanged.

HG.Step 6.2.1: Next, extend t_6 by $v = c_3$, and get (a_3, b_2, c_3). This extended run gives two pairs, (a_3, c_3) and (b_2, c_3) in AP. Hence, $AP'' = \{(a_3, c_3), (b_2, c_3)\}$.

HG.Step 6.2.2: As $|AP''| > |AP'|$, we set $AP' = \{(a_3, c_3), (b_2, c_3)\}$ and $v' = c_3$. This completes the execution of the inner loop. We have found that c_3 is the best value of C to extend t_6.

HG.Step 6.3: $t'_6 = extend(t_6, c_3) = (a_3, b_2, c_3)$.

$$\mathcal{T}' = \{(a_1, b_1, c_1), (a_1, b_2, c_2), (a_2, b_1, c_3), (a_2, b_2, c_1), (a_3, b_1, c_2), (a_3, b_2, c_3)\}.$$

HG.Step 6.4: Update AP by removing the pairs covered by t'_6.

$$\begin{aligned} AP &= AP - \{(a_3, c_3), (b_2, c_3)\} \\ &= \{(a_1, c_3), (a_2, c_2), (a_3, c_1)\} \end{aligned}$$

HG.Step 7: We return to the main IPO procedure with T' containing six runs.

IPO.Step 3.2: The set of runs $\mathcal{T}$ is now the same as the set $\mathcal{T}'$ computed in procedure HG. The set of uncovered pairs U is $\{(a_1, c_3), (a_2, c_2), (a_3, c_1)\}$. We now move to procedure VG for vertical growth.

IPO.Step 3.3: For each uncovered pair in U, we need to add runs to $\mathcal{T}$.

VG.Step 1: We have $m = 6$, $F = C$, $\mathcal{D}(F) = \{c_1, c_2, c_3\}$, $MP = U$, and $\mathcal{T}' = \emptyset$. Recall that m is the number of runs in $\mathcal{T}'$, F denotes the parameter to be used for vertical growth, and MP the set of missing pairs.

VG.Step 2: We will consider each missing pair in MP and repeat the sub steps. Begin with the pair $(A.a_1, C.c_3)$.

VG.Step 2.1: As $\mathcal{T}'$ is empty, hence move to the next step.

VG.Step 2.2: $t = (a_1, *, c_3)$. $\mathcal{T}' = \{(a_1, *, c_3)\}$.

VG.Step 2: Next missing pair: $\{(A.a_2, C.c_2)\}$.

VG.Step 2.1: No run in $\mathcal{T}'$ is of the kind $(*, dc, c_2)$.

VG.Step 2.2: $t = (a_2, *, c_2)$. $\mathcal{T}' = \{(a_1, *, c_3), (a_2, *, c_2)\}$.

VG.Step 2: Next missing pair: $\{(A.a_3, C.c_1)\}$.

VG.Step 2.1: No run in $\mathcal{T}'$ is of the kind $(*, dc, c_1)$.

VG.Step 2.2: $t = (a_3, *, c_1)$. $\mathcal{T}' = \{(a_1, *, c_3), (a_2, *, c_2), (a_3, *, c_1)\}$. We are done with Step 2.

VG.Step 3: Replace the don't cares in $\mathcal{T}'$ and to obtain the following.

$$\mathcal{T}' = \{(a_1, b_2, c_3), (a_2, b_1, c_2), (a_3, b_1, c_1)\}$$

VG.Step 4: Return to procedure IPO with the following set of runs

$$\{(a_1, b_1, c_1), (a_1, b_2, c_2), (a_2, b_1, c_3), (a_2, b_2, c_1), (a_3, b_1, c_2), (a_3, b_2, c_3), (a_1, b_2, c_3), (a_2, b_1, c_2), (a_3, b_1, c_1)\}.$$

This loop is now terminated as there are no more parameters to be processed. The desired covering array, namely $MCA(9, 2^1 3^2, 2)$, is listed below in row–column format. We have replaced a_i and c_i by $i, 1 \le i \le 3$ and b_i by $i, 1 \le i \le 2$. Parameters A, B, and C are labeled as F_1, F_2, and F_3, respectively.

Run	$F_1(A)$	$F_2(B)$	$F_3(C)$
1	1	1	1
2	1	2	2
3	2	1	3
4	2	2	1
5	3	1	2
6	3	2	3
7	1	2	3
8	2	1	2
9	3	1	1

That completes our presentation of an algorithm to generate covering arrays. A detailed analysis of the algorithm has been given by its authors (see the Bibliographic Notes section). To be of any practical use, procedure IPO needs several modifications. Exercise 6.23 lists various types of constraints that are often found in practice and asks you to implement a revised version of procedure IPO.

6.12 Tools

Several commercial and free tools are available for the generation of combinatorial designs. AETG was one of the early tools developed at Bellcore (later Telcordia) and remained as the foremost tool in this area. Subsequently, NIST and Professor Jeff Y. Lei from the University of Texas at Arlington developed another tool named ACTS. This tool is available freely from NIST.

`Tool:` AETG
`Link: http://aetgweb.argreenhouse.com/`
`Description`

The Automatic Efficient Testcase Generator (AETG) was developed at Telcordia Technologies (now Applied Communications Sciences). It is as an online test generation service and can be used to generate *t*-way test configurations for a variety of factor mix. A formal language can be used to specify constraints among factors. While AETG is a commercial tool, its developers have been liberal in offering free short term access for student training.

`Tool:` ACTS
`Link: http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html`
`Description`

ACTS is another freely available tool for generating test configurations. It was developed by Jeff Lie's group in collaboration with NIST. It is available from NIST as a free download. ACTS is a standalone Java application. It can be used from command line or through a simple GUI. It offers several variations of the IPO algorithm described in this chapter to generate test configurations.

SUMMARY

This chapter has covered a well known set of techniques for the generation of tests for software applications. These techniques fall under the general umbrella of *model-based testing* and assist with the generation of tests using combinatorial designs. The primary goal of these techniques is to allow the selection of a small number of tests, or test configurations, from

a potentially huge test input, or test configuration, domain. While this is also the goal of techniques such as equivalence partitioning and boundary value analysis, test generation based on combinatorial designs is a more formal, and often complementary, discipline.

The combinatorial designs covered in this chapter include orthogonal arrays, mixed orthogonal arrays, covering arrays, and mixed-level covering arrays. In addition, we have explained how to generate tests using mutually orthogonal Latin squares. Mixed-level covering arrays seem to be the most popular choice of combinatorial designs amongst software testers. While there exist several tools and methods for the generation of such designs, we have chosen to describe the In-Parameter Order procedure due to its effectiveness in generating near-minimal mixed-level covering designs and simplicity. Several exercises, and a term project, at the end of this chapter are designed to help you to test and further sharpen your understanding of combinatorial design-based software testing.

Exercises

6.1 When and why would an infeasible test case be useful?

6.2 Program P6.1 contains two `if` statements. As explained in Example 6.8, this program contains an interaction fault. Consider a test set T that ensures that each of the two `if` statements is executed at least once and that each of the two conditions ($x == x_1$ `and` $y == y_2$) and ($x == x_2$ `and` $y == y_1$) evaluates to `true` and `false` in some execution on an input from T. (a) Does there exist a test T which will not reveal the interaction fault? (b) Why might the value of z cause the fault to be not revealed even when the fault is triggered?

6.3 Construct an example program that contains one 2-way and one 3-way interaction fault. Generate a test set T that reveals the two interaction faults. Specify any conditions that must be satisfied for T to be able to reveal the faults.

6.4 A set X is known as a *quasigroup* under a binary operation *op* if (a) *op* is defined on the elements of X, (b) there exist unique elements $x, y \in X$ such that for all $a, b \in X$, $a\ op\ x = b$ and $y\ op\ a = b$. Prove that the multiplication table for a quasigroup is a Latin Square. (For answer see pages 16–17 in [142].)

6.5 Construct a multiplication table of non-zero integers modulo 5. For example, in your table, $2 \times 4 = 3$. (a) Is the table you constructed a Latin square? Of what order? (b) Can you generalize this method to construct a Latin square of any order $n > 2$?

6.6 Construct a 4×4 addition table of binary 2-bit strings modulo 2. For example, $01 + 11 = 10$ and $11 + 11 = 00$. Replace each element in the table by its decimal equivalent plus 1. For example, 11 is replaced by 4. Is the resulting table a Latin square? Generalize this method for constructing Latin squares of order n.

6.7 Construct MOLS(7) given $S = \{\star, \circ, \bullet, \times, \triangle, \triangledown\}$. (Note: You may have guessed that it is easier to write a computer program to construct MOLS(n) when n is a prime or a power of prime than to do so by hand, especially for larger values of n.)

6.8 Given three two-valued factors X, Y, and Z, list all minimal sets of factor combinations that cover all pairs of values.

6.9 Let $N(n)$ denote the maximum number of mutually orthogonal Latin squares. What is $N(k)$ for $k = 1, 2, 3$?

6.10 In Example 6.17, we modified the design generated using MOLS so as to satisfy the operating system constraint. However, we did so by altering the values in specific entries so that all pairs are covered. Suggest other ways to resolving the infeasible values when using MOLS to generate test configurations. Discuss advantages and disadvantages of the ways you suggest.

6.11 Example 6.17 illustrates how to obtain a statistical design that covers all interaction pairs in 20 configurations instead of 32.

(a) Enumerate all test configurations by listing the levels for each factor.
(b) How many interactions pairs are covered more than once?
(c) Derive an alternate design that covers all interaction pairs and satisfies the operating system and hardware related constraint.
(d) Is your design better than the design in Example 6.17 in any way?
(e) Can you think of any error in the AGTCS software that might not be discovered by testing against the 20 configurations derived in Example 6.17?

6.12 This exercise is intended to show how constraints on factors could lead to the partitioning of the test configuration problem into two or more simpler problems that are solved independently using the same algorithm.

(a) Design test configurations for testing the AGTCS application described in Example 6.17. Use the PDMOLS procedure to arrive at two sets of test configurations, one for the PC and the other for the Mac. Note that the PDMOLS procedure is to be applied on two sets of factors, one related to the PC and the other to the Mac.
(b) Compare the test configurations derived in (a) with those derived in Example 6.17 in terms of their impact on the test process for AGTCS.

6.13 How many test cases will be generated if the PDMOLS procedure is applied to the GUI problem in Example 6.16?

6.14 Given three factors $x \in \{-1, 1\}$, $y \in \{-1, 1\}$, and $z \in \{0, 1\}$, construct an orthogonal array $OA(N, 3, 2, 2)$. (a) Using the array you have constructed, derive a set of N tests for the program in Example 6.9. (b) Which of the N tests triggers the fault in Program P6.2? (c) What conditions must be satisfied for the fault to be revealed, i.e. propagate to the output of the program? (d) Construct, if possible, non-trivial functions f and g that will prevent the triggered fault to propagate to the output.

6.15 Consider an application iAPP intended to be used from any of three different browsers (Safari, Internet Explorer, and Firefox) running on three operating systems (Windows, Mac OS, and Linux). The application must be able to connect to other devices using any of three connection protocols (LAN, PPP, and ISDN). The application must also be able to send output to a printer that is networked or local, or to the screen.

(a) Identify the factors and levels in the problem mentioned above.
(b) Use Procedure PDMOLS to generate test configurations for iAPP.

6.16 Example 6.22 suggests the use of $OA(54, 5, 3, 3)$ to cover all 3-way interactions. Look up this array on the Internet and design all 54 tests for the pacemaker. Refer to the "CRC Handbook of Combinatorial Design" cited in the Bibliographic Notes section. How many tests are required if the coverage requirement is relaxed to pairwise coverage?

6.17 As indicated in Example 6.22, a total of 54 tests are required to cover all 3-way interactions. (a) What is the minimum number of tests required to test for all 3-way interactions given five parameters each assuming one of three values. (b) Design a minimal covering array $CA(N, 5, 3, 3)$, where N is the number of runs needed.

6.18 A *complete factorial* array of Q factors is a combinatorial object that contains all possible combinations of Q factors. In some cases, a combinatorial array is a minimal size array of a given strength and index. Which of the following arrays is complete factorial? (We have not specified the number of runs. For answers, refer to the handbook mentioned in Exercise 6.16.)

(a) $MA(N, 3^1 5^1, 2)$
(b) $OA(N, 4, 3, 2)$
(c) $MA(N, 2^1 3^1, 2)$
(d) $OA(N, 5, 2, 4)$

6.19 Justify the name "In-Parameter-Order" for procedure IPO.

6.20 Is $MCA(10, 2^1 3^2, 2)$ generated in Example 6.23 using procedure IPO, a minimal array? Is this a mixed orthogonal array?

6.21 In Example 6.23, we assumed $F_1 = A$, $F_2 = B$, and $F_3 = C$. Would the size of the resulting array be different if we were to use the assignment $F_1 = A$, $F_2 = C$, and $F_3 = B$?

6.22 (a) At Step 3 of procedure VG, we can replace the don't care values to maximize the number of higher order tuples. Can you replace the don't care values in Example 6.23, VG.Step 3, so that the number of triples covered is more than the number covered in the array on page 341? (b) What is the minimum array size needed to cover all possible triples of factors A, B, and C?

6.23 *This is a term project.*

(a) Consider the following types of constraints that might exist in a software test environment.
Prohibited pairs
A set of pairs that must not appear in the covering array.
Prohibited higher order tuples
A set of n-order tuples, $n > 2$, must not appear in a covering array of strength 2.
Factor coupling
If factor A is set to r then factor B must be set to s, where r and s belong to, respectively, $\mathcal{D}(A)$ and $\mathcal{D}(B)$. This relationship can be generalized to the following.

If $X(F_1, F_2, \ldots, F_n)$ then $Y_1, Y_2, \ldots, Y_m; n, m \geq 1$.

where X is a relation amongst one or more factors and Ys are constraints on factors. For example, a simple relation exists when $X \in \{<, \leq, =, >, \geq, \neq\}$. Such relations might also exist among more than two factors.
Procedure IPO does not consider any factor related constraints while generating a covering design. Rewrite IPO by including the ability to input and satisfy the constraints listed above. You might solve this problem in an incremental fashion. For example, you might start with modifying IPO so that only the *Prohibited pairs* constraint is included. Then proceed to include the other constraints incrementally.

(b) Think of some practical relations that correspond to X above. Include these in the revised IPO.

(c) Write a Java applet that implements the revised IPO procedure.

(d) If you want to, commercialize your applet and make some money by selling it! watch out for companies that have patented algorithms for the generation of combinatorial designs.

Part III

Test Adequacy Assessment and Enhancement

Techniques to answer "Is my testing adequate?" are introduced in the next two chapters. Chapter 7 presents the foundations of test completeness as defined by Goodenough and Gerhart. This is followed by definitions and illustrations of test adequacy criteria based on the control flow and data flow structure of the program under test.

Chapter 8 is an in-depth presentation of some of the most powerful test adequacy criteria based on program mutation. In each of the two chapters mentioned, we provide several examples to show how errors are detected, or not detected, while enhancing an inadequate test set with reference to an adequacy criterion.

This page is intentionally left blank.

7

Test Adequacy Assessment Using Control Flow and Data Flow

CONTENTS

7.1 Test adequacy: basics
7.2 Adequacy criteria based on control flow
7.3 Concepts from data flow
7.4 Adequacy criteria based on data flow
7.5 Control flow versus data flow
7.6 The "subsumes" relation
7.7 Structural and functional testing
7.8 Scalability of coverage measurement
7.9 Tools

This chapter introduces methods for the assessment of test adequacy and test enhancement. Measurements of test adequacy using criteria based on control flow and data flow are explained. These code based coverage criteria allow a tester to determine how much of the code has been tested and what remains untested.

7.1 Test Adequacy: Basics

7.1.1 What is test adequacy?

Test adequacy refers to the "goodness" of a test set. The "goodness" ought to be measured against a quantitative criterion.

Consider a program P written to meet a set R of functional requirements. We notate such a P and R as (P, R). Let R contain n requirements labeled $R_1, R_2, \ldots, R_n$. Suppose now that a set T containing k tests has been constructed to test P to determine whether or not it meets all the requirements in R. Also, P has been executed against each test in T and has produced correct behavior. We now ask: *Is T good enough?* This question can be stated differently as: *Has P been tested thoroughly?* or as: *Is T adequate?* Regardless of how the question is stated, it assumes importance when one wants to test P thoroughly in the hope that all errors have been discovered and removed when testing is declared complete and the program P declared usable by the intended users.

In the context of software testing, the terms "thorough," "good enough,"and "adequate," used in the questions above, have the same meaning. We prefer the term "adequate" and the question *Is T adequate?* Adequacy is measured for a given test set designed to test P to determine whether or not P meets its requirements. This measurement is done against a given criterion C. A test set is considered adequate with respect to criterion C when it *satisfies* C. The determination of whether or not a test set T for program P satisfies criterion C depends on the criterion itself and is explained later in this chapter.

Functional requirements are concerned with application functionality. Non-functional requirements include characteristics such as performance, reliability, usability.

In this chapter we focus only on *functional* requirements, testing techniques to validate non-functional requirements are dealt with elsewhere.

Example 7.1 Consider the problem of writing a program named `sumProduct` that meets the following requirements:

R_1: Input two integers, say x and y, from the standard input device.

$R_{2.1}$: Find and print to the standard output device the sum of x and y if $x < y$.

$R_{2.2}$: Find and print to the standard output device the product of x and y if $x \geq y$.

Suppose now that the test adequacy criterion C is specified as follows:

C: *A test T for program (P, R) is considered adequate if for each requirement r in R there is at least one test case in T that tests the correctness of P with respect to r.*

It is obvious that $T = \{t : < x = 2, y = 3 >\}$ is inadequate with respect to C for program `sumProduct`. The only test case t in T tests R_1 and $R_{2.1}$, but not $R_{2.2}$.

Test adequacy is measured against a coverage domain. Such a domain could be based on requirements or elements of the code. A test set that is adequate only with respect to a requirements based coverage domain is likely to be weaker than that adequate with respect to both requirements and code-based coverage domains.

7.1.2 Measurement of test adequacy

Adequacy of a test set is measured against a finite set of elements. Depending on the adequacy criterion of interest, these elements are derived from the requirements or from the program under test. For each adequacy criterion C, we derive a finite set known as the *coverage domain* and denoted as C_e.

A criterion C is a *white-box* test adequacy criterion if the corresponding coverage domain C_e depends solely on program P under test. A criterion C is a *black-box* test adequacy criterion if the corresponding coverage domain C_e depends solely on requirements R for the program P under test. All other test adequacy criteria are of a mixed nature and not considered in this chapter. This chapter introduces several white-box test adequacy criteria that are based on the flow of control and the flow of data within the program under test.

An adequacy criterion derived solely from a code-based coverage domain is a white-box criterion. That derived from a requirements-based coverage domain is a black box. criterion.

Suppose that it is desired to measure the adequacy of T. Given that C_e has $n \geq 0$ elements, we say that T *covers* C_e if for each element e' in C_e there is at least one test case in T that tests e'. T is considered adequate with respect to C if it covers all elements in the coverage domain. T is considered inadequate with respect to C if it covers k elements of C_e where $k < n$. The fraction k/n is a measure of the extent to which T is adequate with respect to C. This fraction is also known as the *coverage* of T with respect to C, P, and R.

The determination of when an element e is considered *tested* by T depends on e and P and is explained below through examples.

Example 7.2 Consider the program P, test T, and adequacy criterion C of Example 7.1. In this case the finite set of elements C_e is the set $\{R_1, R_{2.1}, R_{2.2}\}$. T covers R_1 and $R_{2.1}$ but not $R_{2.2}$. Hence T is not adequate with respect to C. The coverage of T with respect to C, P, and R is 0.66. Element $R_{2.2}$ is not tested by T whereas the other elements of C_e are tested.

Example 7.3 Next let us consider a different test adequacy criterion which is referred to as the *path coverage* criterion.

C: *A test T for program (P, R) is considered adequate if each path in P is traversed at least once.*

Given the requirements in Example 7.1 let us assume that P has exactly two paths, one corresponding to condition $x < y$ and the other to $x \geq y$. Let us refer to these two paths as p_1 and p_2, respectively. For the given adequacy criterion C we obtain the coverage domain C_e to be the set $\{p_1, p_2\}$.

To measure the adequacy of T of Example 7.1 against C, we execute P against each test case in T. As T contains only one test for which $x < y$, only the path p_1 is executed. Thus the coverage of T with respect to C, P, and R is 0.5 and hence T is not adequate with respect to C. We also say that p_2 is not tested.

A code-based coverage domain is derived through the analysis of the program under test.

In Example 7.3, we assumed that P contains exactly two paths. This assumption is based on a knowledge of the requirements. However, when the coverage domain must contain elements from the code, these elements must be derived by program analysis and not by an examination of its requirements. Errors in the program and incomplete or incorrect requirements might cause the program, and hence the coverage domain, to be different from what one might expect.

Example 7.4 Consider the following program written to meet the requirements specified in Example 7.1; the program is obviously incorrect.

Program P7.1

```
begin
  int x, y;
   input (x, y);
   sum=x+y;
   output (sum);
end
```

The above program has exactly one path which we denote as p_1. This path traverses all statements. Thus, to evaluate any test with respect to criterion C of Example 7.3, we obtain the coverage domain C_e to be $\{p_1\}$. It is easy to see that C_e is covered when P is executed against the sole test in T of Example 7.1. Thus T is adequate with respect to P even though the program is incorrect.

Program P7.1 has an error that is often referred to as a "missing path" or a"missing condition" error. A correct program that meets the requirements of Example 7.1 follows.

Program P7.2

```
begin
  int x, y;
  input (x, y);
  if(x<y)
  then
     output (x+y);
  else
     output (x*y);
end
```

A missing condition leads to a missing path error. There is no guarantee that such an error will be detected by a test set adequate with respect to any white box or black box criteria that has a finite coverage domain.

This program has two paths, one of which is traversed when $x < y$ and the other when $x \geq y$. Denoting these two paths by p_1 and p_2 we obtain the coverage domain given in Example 7.3. As mentioned earlier, test T of Example 7.1 is not adequate with respect to the path coverage criterion.

The above example illustrates that an adequate test set might not reveal even the most obvious error in a program. This does not diminish in any way the need for the measurement of test adequacy. The next section explains the use of adequacy measurement as a tool for test enhancement.

7.1.3 Test enhancement using measurements of adequacy

Quantitatively measured inadequacy of a test set is an opportunity for its enhancement.

While a test set adequate with respect to some criterion does not guarantee an error-free program, an inadequate test set is a cause for worry. Inadequacy with respect to any criterion often implies deficiency. Identification of this deficiency helps in the enhancement of the inadequate test set. Enhancement in turn is also likely to test the program in ways it has not been tested before such as testing untested portion, or testing the features in a sequence different from the one used previously. Testing the program differently than before raises the possibility of discovering any uncovered errors.

Example 7.5 Let us reexamine test T for P7.2 in Example 7.4. To make T adequate with respect to the path coverage criterion, we need to add a test that covers p_2. One test that does so is $\{< x = 3, y = 1 >\}$. Adding this test to T and denoting the expanded test set by T', we get

$$T' = \{< x = 2, y = 3 >, < x = 3, y = 1 >\}.$$

When P7.2 is excuted against the two tests in T', both paths p_1 and p_2 are traversed. Thus T' is adequate with respect to the path coverage criterion.

Given a test set T for program P, test enhancement is a process that depends on the test process employed in the organization. For each new test added to T, P needs to be executed to determine its behavior. An erroneous behavior implies the existence of an error in P and will likely lead to debugging of P and the eventual removal of the error. However, there are several procedures by which the enhancement could be carried out. One such procedure follows.

```
Procedure for Test Enhancement Using Measurements
of Test Adequacy.
```

Step 1 Measure the adequacy of T with respect to the given criterion C. If T is adequate then go to Step 3, otherwise execute the next step. Note that during adequacy measurement we will be able to determine the uncovered elements of C_e.

Step 2 For each uncovered element $e \in C_e$, do the following until e is covered or is determined to be infeasible.

2.1 Construct a test t that covers e or will likely cover e.

2.2 Execute P against t.

2.2.1 If P behaves incorrectly then we have discovered the existence of an error in P. In this case t is added to T, the error is removed from P and this procedure gets repeated from the beginning.

2.2.2 If P behaves correctly and e is covered then t is added to T, otherwise it is the tester's option whether to ignore t or to add it to T.

Step 3 Test enhancement is complete.

```
End of procedure
```

Adequacy-based test enhancement is a cycle in which a test set is enhanced by an examination of which parts of the corresponding coverage domain are not covered. New tests added are designed to cover the uncovered portions of the coverage domain and thus enhance the test set.

Figure 7.1 shows a sample test construction-enhancement cycle. The cycle begins with the construction of a non-empty set T of test cases. These tests cases are constructed from the requirements of program P under test. P is then executed against all test cases. If P fails on one or more of the test cases then it ought to be corrected by first finding and then removing the source of the failure. The adequacy of T is measured with respect to a suitably selected adequacy criterion C after P is found to behave satisfactorily on all elements of T. This construction-enhancement cycle is considered complete if T is found adequate with respect to C. If not, then additional test cases are constructed in an attempt to remove the deficiency.

Figure 7.1 A test construction-enhancement cycle.

The construction of these additional test cases once again makes use of the requirements that *P* must meet.

Example 7.6 Consider the following program intended to compute x^y given integers x and y. For $y < 0$ the program skips the computation and outputs a suitable error message.

Program P7.3

```
begin
  int  x, y;
  int  product, count;
  input (x, y);
  if (y≥0)  {
     product=1; count=y;
    while (count>0)  {
       product=product*x;
       count=count-1;
```

```
        }//  End of while
    output(product);
   }//  End of true part of if
   else
      output ("Input does not match its
         specification.");
   end
```

Next, consider the following test adequacy criterion.

C: *A test T is considered adequate if it tests P7.2 for at least one zero and one non-zero value of each of the two inputs* `x` *and* `y`.

The coverage domain for C can be determined using C alone and without any inspection of P7.3. For C, we get $C_e = \{x = 0, y = 0, x \neq 0, y \neq 0\}$. Again, one can derive an adequate test set for 7.6 by a simple examination of C_e. One such test set is

$$T = \{< x = 0, y = 1 >, < x = 1, y = 0 >\}.$$

In this case, we need not apply the enhancement procedure given above. Of course, P7.3 needs to be executed against each test case in T to determine its behavior. For both tests it generates the correct output which is 0 for the first test case and 1 for the second. Note that T might well be generated without reference to any adequacy criterion.

A black-box coverage criterion does not require an examination of the program under test in order to generate new tests.

Example 7.7 Criterion C of Example 7.6 is a *black-box* coverage criterion as it does not require an examination of the program under test for the measurement of adequacy. Let us consider the path coverage criterion defined in Example 7.3. An examination of P7.3 reveals that it has an indeterminate number of paths due to the presence of a while loop. The number of paths depends on the value of `y` and hence that of `count`. Given that `y` is any non-negative integer, the number of paths can be arbitrarily large. This simple analysis of paths in P7.3 reveals that we cannot determine the coverage domain for the path coverage criterion.

The usual approach in such cases is to simplify C and reformulate it as follows.

C′: *A test T is considered adequate if it tests all paths. In case the program contains a loop, then it is adequate to traverse the loop body zero times and once.*

Figure 7.2 Control flow graph of P7.6.

The modified path coverage criterion leads to $C'_e = \{p_1, p_2, p_3\}$. The elements of C_e are enumerated below with respect to Figure 7.2.

A possible coverage domain consists of all paths in a program. Such a domain is made finite by assuming that each loop body in the program is skipped on at least one arrival at the loop and it is executed at least once on another arrival.

p_1: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 9]$; corresponds to $y \geq 0$ and loop body traversed zero times.

p_2: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 5 \rightarrow 7 \rightarrow 9]$; corresponds to $y \geq 0$ and loop body traversed once.

p_3: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 8 \rightarrow 9]$; corresponds to $y < 0$ and the control reaches the `output` statement without entering the body of the `while` loop.

The coverage domain for C' and P7.3 is $\{p_1, p_2, p_3\}$. Following the test enhancement procedure, we first measure the adeqacy of T with respect to C'. This is done by executing P against each element of T and determining which elements in C'_e are covered and which ones are not. As T does not contain any test with $y < 0$, p_3 remains uncovered. Thus the coverage of T with respect to C' is $2/3 = 0.66$.

Moving on to Step 2, we attempt to construct a test aimed at covering p_3. Any test case with $y < 0$ will cause p_3 to be traversed.

Let us use the test case $t :< x = 5, y = -1 >$. When P is executed against t, indeed path p_3 is covered and P behaves correctly. We now add t to T. The loop in Step 2 is now terminated as we have covered all feasible elements of C'_e. The enhanced test set is

$$T = \{< x = 0, y = 1 >, < x = 1, y = 0 >, < x = 5, y = -1 >\}.$$

Note that T is adequate with respect to C', but is Program P7.3 correct?

7.1.4 Infeasibility and test adequacy

An element of a coverage domain, e.g., a path, might be infeasible to cover. This implies that there exists no test case that covers such an element.

An element of the coverage domain is infeasible if it cannot be covered by any test in the input domain of the program under test. In general, it is not possible to write an algorithm that would analyze a given program and determine if a given element in the coverage domain is feasible or not. Thus it is usually the tester who determines whether or not an element of the coverage domain is infeasible.

Feasibility can be demonstrated by executing the program under test against a test case and showing that indeed the element under consideration is covered. However, infeasibility cannot be demonstrated by program execution against a finite number of test cases. Nevertheless, as in Example 7.8, simple arguments can be constructed to show that a given element is infeasible. For more complex programs, the problem of determining infeasibility could be difficult. Thus one might fail while attempting to cover e to enhance a test set by executing P against t.

The conditions that must hold for a test case t to cover an element e of the coverage domain depend on e and P. These conditions are derived later in this chapter when we examine various types of adequacy criteria.

Example 7.8 In this example, we illustrate how an infeasible path occurs in a program. The following program inputs two integers `x` and `y` and computes `z`.

Program P7.4

```
1   begin
2     int x, y;
3     int z;
4      input (x, y); z=0;
5     if( x<0 and y<0){
6       z=x*x;
7       if (y≥ 0) z=z+1;
```

```
8       }
9     else
10       z=x*x*x;
11    output(z);
12   end
```

The path coverage criterion leads to $C_e = \{p_1, p_2, p_3\}$. The elements of C_e are enumerated below with respect to Figure 7.3.

p_1: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 8 \rightarrow 9]$; corresponds to conditions $x < 0$ and $y < 0$ and $y \geq 0$ evaluating to true.

p_2: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 8 \rightarrow 9]$; corresponds to conditions $x < 0$ and $y < 0$ evaluating to true and $y \geq 0$ to false.

p_3: $[1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 9]$; corresponds to $x < 0$ and $y < 0$ evaluating to false.

It is easy to check that path p_1 is infeasible and cannot be traversed by any test case. This is because when control reaches node 5,

For short programs one might be able to easily categorize a path to be feasible or not. However, for large applications doing so might be a humanly impossible task. The general problem of determining path infeasibility is unsolvable.

Figure 7.3 Control flow graph of P7.4.

condition $y \geq 0$ is false and hence control can never reach node 6. Thus any test adequate with respect to the path coverage criterion for P7.4 will only cover p_2 and p_3.

In the presence of one or more infeasible elements in the coverage domain, a test is considered adequate when all feasible elements in the domain have been covered. This also implies that in the presence of infeasible elements, adequacy is achieved when coverage is less than 1.

Infeasible elements in the coverage domain make it harder for testers to generate adequate tests.

Infeasible elements arise for a variety of reasons discussed in subsequent sections. While programmers might not be concerned with infeasible elements, testers attempting to obtain code coverage are. Prior to test enhancement, a tester usually does not know which elements of a coverage domain are infeasible. It is only during an attempt to construct a test case to cover an element that one might realize the infeasibility of an element. For some elements, this realization might come after several failed attempts. This may lead to frustration on the part of the tester. The testing effort spent on attempting to cover an infeasible element might be considered wasteful. Unfortunately there is no automatic way to identify all infeasible elements in a coverage domain derived from an arbitrary program. However, careful analysis of a program usually leads to a quick identification of infeasible elements. We return to the topic of dealing with infeasible elements later in this chapter.

7.1.5 Error detection and test enhancement

Tests designed based only on requirements can often be enhanced significantly by an examination of code that remains untested. Such enhancement has the potential of revealing errors that might have otherwise remained uncovered.

The purpose of test enhancement is to determine test cases that test the untested parts of a program. Even the most carefully designed tests based exclusively on requirements can be enhanced. The more complex the set of requirements, the more likely it is that a test set designed using requirements is inadequate with respect to even the simplest of various test adequacy criteria.

During the enhancement process, one develops a new test case and executes the program against it. Assuming that this test case exercises the program in a way it has not been exercised before, there is a chance that an error present in the newly tested portion of the program is revealed. In general, one cannot determine how probable or improbable it is to reveal an error through test enhancement. However, a carefully designed and executed test enhancement process is often useful in locating program errors.

Example 7.9 A program to meet the following requirements is to be developed.

R_1: Upon start the program offers the following three options to the user:

- Compute x^y for integers x and $y \geq 0$.
- Compute the factorial of integer $x \geq 0$.
- Exit.

$R_{1.1}$: If the "Compute x^y" option is selected then the user is asked to supply the values of x and y, x^y is computed and displayed. The user may now select any of the three options once again.

$R_{1.2}$: If the "Compute factorial x" option is selected then the user is asked to supply the value of x and factorial of x is computed and displayed. The user may now select any of the three options once again.

$R_{1.3}$: If the "Exit" option is selected the program displays a goodbye message and exits.

Consider the following program written to meet the above requirements.

Program P7.5

```
1   begin
2    int x, y;
3    int  product, request;
4     #define exp=1
5     #define fact=2
6     #define exit=3
7     get_request (request); // Get user request
     (one of three possibilities).
8     product=1; // Initialize product.
9     // Set up the loop to accept and execute
       requests.
10   while (true) {
11    //  Process the "exponentiation" request.
12     if  (request == exp){
13         input (x, y); count=y;
14        while (count > 0){
15          product=product*x; count=count-1;
16        }
17   } //  End of processing the "exponentiation"
             request.
18    //  Process "factorial" request.
19     else if  (request == fact){
```

```
20          input (x); count=x;
21         while  (count >0){
22           product=product*count;  count=count-1;
23         }
24      }  // End of  processing the "factorial" request.
25   //   Process  "exit" request.
26   elseif (request == exit){
27       output("Thanks for using this program. Bye!");
             break; // Exit the loop.
28   }   //   End of   if.
29    output(product); // Output the value of exponential
      or factorial.
30     get_request (request); // Get user request once
       again and jump to loop begin.
31     }
32   end
```

Suppose now that the following set containing three tests labeled t_1, t_2, and t_3 has been developed to test P7.5.

$$
\begin{aligned}
T = \{\ & t_1 : < request = 1, x = 2, y = 3, request = 3 > \\
& t_2 : < request = 2, x = 4, request = 3 > \\
& t_3 : < request = 3 > \\
\}\ &
\end{aligned}
$$

This program is executed against the three tests in the sequence they are listed above. The program is launched once for each input. For the first two of the three requests the program correctly outputs 8 and 24, respectively. The program exits when executed against the last request. This program behavior is correct and hence one might conclude that the program is correct. It will not be difficult for you to believe that this conclusion is incorrect.

Let us evaluate T against the path coverage criterion described earlier in Example 7.9. Before you proceed further, you might want to find a few paths in P7.5 that are not covered by T.

A path-based coverage domain consists of all paths in a program such that all loops are executed zero and once. Such executions could be in the same or different runs of the program.

The coverage domain consists of all paths that traverse each of the three loops zero *and* once in the same or different executions of the program. We leave the enumeration of all such paths in P7.5 as an exercise. For this example, let us consider the path that begins execution at line 1, reaches the outermost `while` at line 10, then the first `if` at line 12, followed by the statements that compute the factorial starting at line 20, and then the code to compute the exponential starting at line 13. For this example, we do not care what happens after the exponential has been computed and output.

Our tricky path is traversed when the program is launched and the first input request is to compute the factorial of a number, followed by a request to compute the exponential. It is easy to verify that the sequence of requests in T do not exercise p. Therefore T is inadequate with respect to the path coverage criterion.

To cover p we construct the following test:

$$T' = \{< request = 2, x = 4 >, < request = 1, x = 2, y = 3 >, < request = 3 >\}$$

When the values in T' are input to our example program in the sequence given, the program correctly outputs 24 as the factorial of 4 but incorrectly outputs 192 as the value of 2^3. This happens because T' traverses our tricky path which makes the computation of the exponentiation begin without initializing `product`. In fact the code at line 14 begins with the value of `product` set to 24.

Note that in our effort to increase the path coverage we constructed T'. Execution of the test program on T' did cover a path that was not covered earlier and revealed an error in the program.

7.1.6 Single and multiple executions

In Example 7.9, we constructed two test sets T and T'. Notice that both T and T' contain three tests one for each value of variable `request`. Should T be considered a single test or a set of three tests? The same question applies also to T'. The answer depends on how the values in T are input to the test program. In our example, we assumed that all three tests, one for each value of `request`, are input in a sequence during *a single execution of the test program*. Hence we consider T as a test set containing one test case and write it as follows:

An test sequence input in one run of the program might not reveal an error that could otherwise be revealed when the same test is input across multiple runs. Of course, this assume that the test can be partitioned so that individual portions are input across different runs. Note that each run starts the program fro its initial state.

$$T = \left\{ \begin{array}{lll} t_1: & << request = 1, x = 2, y = 3 > & \rightarrow \\ & < request = 2, x = 4 > & \rightarrow \quad < request = 3 >> \end{array} \right\}$$

Note the use of the outermost angular brackets to group all values in a test. Also, the right arrow ($\rightarrow$) indicates that the values of variables are changing in the same execution. We can now rewrite T' also in a way similar to T. Combining T and T', we get a set T'' containing two test cases written as follows:

$$T'' = \left\{ \begin{array}{lll} t_1: & << request = 1, x = 2, y = 3 > & \rightarrow \quad < request = 2, x = 4 > \\ & \rightarrow \quad < request = 3 >> & \\ t_2: & << request = 2, x = 4 > & \rightarrow \\ & < request = 1, x = 2, y = 3 > & \rightarrow \quad < request = 3 >> \end{array} \right\}$$

Test set T'' contains two test cases, one that came from T and the other from T'. You might wonder why so much fuss regarding whether something is a test case or a test set. In practice, it does not matter. Thus you may consider T as a set of three test cases or simply as a set of one test case. However, we do want to stress the point that distinct values of all program inputs may be input in separate runs or in the same run of a program. Hence a set of test cases might be input in a single run or in separate runs.

Programs that offer a GUI for user interaction are likely to be tested across multiple test cases in a single run.

In older programs that were not based on Graphical User Interfaces (GUI), it is likely that all test cases were executed in separate runs. For example, while testing a program to sort an array of numbers, a tester usually executed the sort program with different values of the array in each run. However, if the same sort program is "modernized" and a GUI added for ease of use and marketability, one may test the program with different arrays input in the same run.

In the next section, we introduce various criteria based on the flow of control for the assessment of test adequacy. These criteria are applicable to any program written in a procedural language such as C. The criteria can also be used to measure test adequacy for programs written in object oriented languages such as Java and C++. Such criteria include plain method coverage as well as method coverage within context. The criteria presented in this chapter can also be applied to programs written in low level languages such as an assembly language. We begin by introducing *control flow based test adequacy criteria*.

7.2 Adequacy Criteria Based on Control Flow

7.2.1 Statement and block coverage

Programs in a procedural language consists of a set of functions, each function being a sequence of statements. In OO languages the programs are structured into classes where each class often contains multiple methods (functions).

Any program written in a procedural language consists of a sequence of statements. Some of these statements are declarative, such as the `#define` and `int` statements in C, while others are executable, such as the assignment, `if` and `while` statements in C and Java. Note that a statement such as

```
if count=10;
```

could be considered declarative because it declares the variable `count` to be an integer. This statement could also be considered as executable because it assigns 10 to the variable `count`. It is for this reason that in C we consider all declarations as executable statements when defining test adequacy criteria based on the flow of control.

Recall from Chapter 1 that a *basic block* is a sequence of consecutive statements that has exactly one entry point and one exit point. For any procedural language, adequacy with respect to the statement coverage and block coverage criteria are defined as follows.

Statement Coverage:
The statement coverage of T with respect to (P, R) is computed as $|S_c|/(|S_e| - |S_i|)$, where S_c is the set of statements covered, S_i the set of unreachable statements, and S_e is the coverage domain consisting of the set of statements in the program. T is considered adequate with respect to the statement coverage criterion if the statement coverage of T with respect to (P, R) is 1.

Block Coverage:
The block coverage of T with respect to (P, R) is computed as $|B_c|/(|B_e| - |B_i|)$, where B_c is the set of blocks covered, B_i the set of unreachable blocks, and B_e the blocks in the program, i.e. the block coverage domain. T is considered adequate with respect to the block coverage criterion if the block coverage of T with respect to (P, R) is 1.

In the above definitions, the coverage domain for statement coverage is the set of all statements in the program under test. Similarly, the coverage domain for block coverage is the set of all basic blocks in the program under test. Note that we use the term "unreachable" to refer to statements and blocks that fall on an infeasible path.

A rather simple test adequacy criterion is based on statement coverage. In this case the coverage domain consists of all statements in the program under test. Alternately, the coverage domain could also be a set of basic blocks in the program under test.

The next two examples explain the use of the statement and block coverage criteria. In these examples we use line numbers in a program to refer to a statement. For example, the number 3 in S_e for 7.1.2 refers to the statement on line 3 of this program, i.e. to the `input (x, y)` statement. We will refer to blocks by block numbers derived from the flow graph of the program under consideration.

Example 7.10 The coverage domain corresponding to statement coverage for P7.4 is given below.

$S_e = \{2, 3, 4, 5, 6, 7, 7b, 9, 10\}$

Here we denote the statement $z = z + 1$; as 7b. Consider a test set T_1 that consists of two test cases against which P7.4 has been executed.

$T_1 = \{t_1 :< x = -1, y = -1 >,\ t_2 :< x = 1, y = 1 >\}$

Statements 2, 3, 4, 5, 6, 7, and 10 are covered upon the execution of P against t_1. Similarly, the execution against t_2 covers statements

2, 3, 4, 5, 9, and 10. Neither of the two tests covers statement 7b which is unreachable as we can conclude from Example 7.8. Thus we obtain $|S_c| = 8$, $|S_i| = 1$, $|S_e| = 9$. The statement coverage for T is $8/(9-1) = 1$. Hence we conclude that T is adequate for (P, R) with respect to the statement coverage criterion.

Some blocks in a coverage domain might be infeasible. Such infeasibility is usually discovered manually.

Example 7.11 The five blocks in P7.4 are shown in Figure 7.4. The coverage domain for the block coverage criterion $B_e = \{1, 2, 3, 4, 5\}$. Consider now a test set T_2 containing three tests against which P7.4 has been executed.

$$T_2 = \left\{ \begin{array}{lll} t_1: & < x = -1 & y = -1 > \\ t_2: & < x = -3 & y = -1 > \\ t_3: & < x = -1 & y = -3 > \end{array} \right\}$$

Blocks 1, 2, and 5 are covered when the program is executed against test t_1. Tests t_2 and t_3 also execute exactly the same set of blocks. For T_2 and P7.4, we obtain $|B_e| = 5$, $|B_c| = 3$, and $|B_i| = 1$. The block coverage can now be computed as $3/(5-1) = 0.75$. As the block coverage is less than 1, T_2 is not adequate with respect to the block coverage criterion.

Figure 7.4 Control flow graph of 7.8. Blocks are numbered 1 through 5. The shaded block 3 is infeasible because the condition in block 2 will never be true.

It is easy to check that the test set of Example 7.10 is indeed adequate with respect to the block coverage criterion. Also, T_2 can be made adequate with respect to the block coverage criterion by the addition of test t_2 from T_1 in the previous example.

The formulae given in this chapter for computing various types of code coverage yield a coverage value between 0 and 1. However, while specifying a coverage value, one might instead use percentages. For example, a statement coverage of 0.65 is the same as 65% statement coverage.

Coverage values are generally computed as a ratio of the number of items covered to the number of feasible items in the coverage domain. Hence, this ratio is between 0 and 1. It is rare to find a ratio of 1, i.e. 100% statement coverage when testing large and complex applications.

7.2.2 Conditions and decisions

To understand the remaining adequacy measures based on control flow, we need to know what exactly constitutes a condition and a decision. Any expression that evaluates to `true` or `false` constitutes a condition. Such an expression is also known as a *predicate*. Given that A, B, and D are Boolean variables, and x and y are integers, A, $x > y$, A `or` B, A `and` $(x < y)$, (A `and` B) `or` (A `and` D) `and` ($\neg D$), (A `xor` B) `and` $(x \geq y)$, are all conditions. In these examples, `and`, `or`, `xor`, and $\neg$ are known as Boolean, or logical, operators. Note that in programming language C, x and $x+y$ are valid conditions, and the constants 1 and 0 correspond to, respectively, `true` and `false`.

A simple predicate (or a condition) is made up of a single Boolean variable (possibly with a negation operator) or of a single relational operator. A compound predicate is a concatenation of two or more simple predicates obtained using the Boolean operators AND, OR, and XOR.

Simple and compound conditions: A condition could be *simple* or *compound*. A simple condition does not use any Boolean operators except for the $\neg$ operator. It is made up of variables and at most one relational operator from the set $\{<, \leq, >, \geq, ==, \neq\}$. A compound condition is made up of two or more simple conditions joined by one or more Boolean operators. In the above examples, A as well as $x > y$ are two simple conditions, while the others are compound. Simple conditions are also referred to as *atomic* or *elementary* conditions because they cannot be parsed any further into two or more conditions. Often, the term *condition* refers to a compound condition. In this book, we will use "condition" to mean any simple or compound condition.

A condition (or a predicate) is used as a decision. The outcome of evaluating a condition is either true or false. A condition involving one or more functions might fail to evaluate in case the function does not terminate appropriately, as for example when it causes the computer to crash.

Conditions as decisions: Any condition can serve as a decision in an appropriate context within a program. As shown in Figure 7.5, most high level languages provide `if`, `while`, and `switch` statements to serve as contexts for decisions. Whereas an `if` and a `while` contains exactly one decision, a `switch` may contain more.

A decision can have three possible outcomes, `true`, `false`, and `undefined`. When the condition corresponding to a decision evaluates to `true` or `false`, the decision to take one or the other path is taken. In the

if (A) *task 1 if A is true;* *else* *task 2 if A is false;* *(a)*	*while(A)* *task while A is true;* *(b)*	*switch (e)* *task 1 for e=e1* *else* *task 2 for e=e2* ⋮ *else* *task n for e=en* *else* *default task* *(c)*

Figure 7.5 Decisions arise in several contexts within a program. Three common contexts for decisions in a C or a Java program are (a) `if`, (b) `while`, and (c) `switch` statements. Note that the `if` and `while` statements force the flow of control to be diverted to one of two possible destinations while the `switch` statement may lead the control to one or more destinations.

While a condition used in an if or a looping statement leads the control to follow one of two possible paths, an expression in a switch statement may cause the control to follow one of several paths. A switch statement generally uses expressions that evaluate to a value from a set of multiple values. Of course, it is possible for a switch statement to use a condition. For Fortran programmers, the condition in an if statement may cause the control to select one of three possible paths.

case of a `switch` statement, one of several possible paths gets selected and the control flow proceeds accordingly. However, in some cases the evaluation of a condition might fail in which case the corresponding decision's outcome is undefined.

Example 7.12 Consider the following sequence of statements.

Program P7.6

```
1   bool foo(int a_parameter){
2   while (true) {        //  An infinite loop.
3       a_parameter=0;
4       }
5   }        //  End of function foo().
⋮
6   if(x<y and  foo(y)){        // foo() does not terminate.
7       compute(x,y);
⋮
```

The condition inside the `if` statement on line 6 will remain undefined because the loop at lines 2–4 will never end. Thus the decision on line 6 evaluates to `undefined`.

Coupled conditions: There is often the question of how many simple conditions are there in a compound condition. For example, $C = (A \texttt{ and } B) \texttt{ or } (C \texttt{ and } A)$ is a compound condition. Does C

contain three or four simple conditions? Both answers are correct depending on one's point of view. Indeed, there are three distinct conditions A, B, and C. However, the answer is four when one is interested in the number of occurrences of simple conditions in a compound condition. In the example expression above, the first occurrence of A is said to be *coupled* to its second occurrence.

A condition is considered coupled when it uses a simple predicate more than once.

Conditions within assignments: Conditions may occur within an assignment statement as in the following examples.

1 $a = x < y$; // A simple condition assigned to a Boolean variable a.
2 $x = p$ `or` q; // A compound condition assigned to a Boolean variable x.
3 $x = y + z * s$; `if` (x)... // Condition `true` if $x = 1$, `false` otherwise.
4 $a = x < y$; $x = a * b$; // a is used in a subsequent expression for x, but not as a decision.

While predicates are generally used in the context of if and looping statements, most languages allow them to be used on the right side of an assignment statement.

A programmer might want a condition to be evaluated before it is used as a decision in a selection or a loop statement, as in the examples above. Strictly speaking, a condition becomes a decision only when it is used in the appropriate context such as within an `if` statement. Thus, in the example at line 4, $x < y$ does not constitute a decision and neither does $a * b$. However, as we shall see in the context of the MC/DC coverage, a decision is not synonymous with a branch point such as that created by an `if` or a `while` statement. Thus, in the context of the MC/DC coverage, the conditions on lines 1, 2, and the first one on line 4 are all decisions too!

7.2.3 Decision coverage

Decision coverage is also known as *branch decision coverage*. A decision is considered *covered* if the flow of control has been diverted to all possible destinations that correspond to this decision, i.e. all outcomes of the decision have been taken. This implies that, for example, the expression in the `if` or `while` statement has evaluated to `true` in some execution of the program under test and to `false` in the same or another execution.

A decision in a program, such as in an if statement, is considered covered if the flow of control has been diverted to all possible destinations during one or more runs of the program under test. An decision in a switch statement might lead to more than two possible destinations.

A decision implied by the `switch` statement is considered covered if during one or more executions of the program under test the flow of control has been diverted to all possible destinations. Covering a decision within a program might reveal an error that is not revealed by covering all statements and all blocks. The next example illustrates this fact.

Example 7.13 To illustrate the need for decision coverage, consider P7.7. This program inputs an integer x, and if necessary, transforms it into a positive value before invoking function `foo-1` to compute the output z. However, as indicated, this program has an error. Assume that as per its requirements, the program must compute z using `foo-2` when $x \geq 0$. Now consider the following test set T for this program.

$T = \{t_1 : < x = -5 >\}$

It is easy to verify that when P7.7 is executed against the sole test case in T, all statements and all blocks in this program are covered. Hence T is adequate with respect to both the statement and the block coverage criteria. However, this execution does not force the condition inside the `if` to be evaluated to `false` thus avoiding the need to compute z using `foo-2`. Hence T does not reveal the error in this program.

Program P7.7

```
1   begin
2     int  x, z;
3     input (x);
4     if(x<0)
5          x=-x;
6          z=foo-1(x);
7     output(z);      ←  There should have been an  else
                            clause before this statement.
8   end
```

Suppose that we add a test case to T to obtain an enhanced test set T'.

$T' = \{t_1 : < x = -5 >, t_2 : < x = 3 >\}$

Decision coverage aids in testing if decisions in a program are correctly formulated and located.

When P7.7 is executed against all tests in T', all statements and blocks in the program are covered. In addition, the sole decision in the program is also covered because condition $x < 0$ evaluates to `true` when the program is executed against t_1 and to `false` when executed against t_2. Of course, control is diverted to the statement at line 6 without executing line 5. This causes the value of z to be computed using `foo-1` and not `foo-2` as required. Now, if `foo-1(3)` $\neq$ `foo-2(3)` then the program will give an incorrect output when executed against test t_2.

The above example illustrates how decision coverage might help a tester discover an incorrect condition and a missing statement

by forcing the coverage of a decision. As you may have guessed, covering a decision does not necessarily imply that an error in the corresponding condition will always be revealed. As indicated in the example above, certain other program dependent conditions must also be true for the error to be revealed. We now formally define adequacy with respect to the decision coverage.

Decision coverage:

The decision coverage of T with respect to (P, R) is computed as $|D_c|/(|D_e| - |D_i|)$, where D_c is the set of decisions covered, D_i the set of infeasible decisions, and D_e the set of decisions in the program, i.e. the decision coverage domain. T is considered adequate with respect to the decision coverage criterion if the decision coverage of T with respect to (P, R) is 1.

The coverage domain in decision coverage is the set of all possible decisions in the program under test.

The domain of decision coverage consists of all decisions in the program under test. Note that each `if` and each `while` contribute to one decision whereas a `switch` may contribute to more than one. For the program in Example 7.13, the decision coverage domain is $D_e = \{x < 0\}$ and hence $|D_e| = 1$.

7.2.4 Condition coverage

A decision can be composed of a simple condition such as $x < 0$, or of a more complex condition, such as $((x < 0 \texttt{ and } y < 0) \texttt{ or } (p \geq q))$. Logical operators `and`, `or`, and `xor` connect two or more simple conditions to form a *compound* condition. In addition, ¬ (pronounced as 'not') is a unary logical operator that negates the outcome of a condition.

Coverage of a decision does not necessarily imply the coverage of the corresponding condition. This is so when the condition is compound.

A simple condition is considered covered if it evaluates to `true` and `false` in one or more executions of the program in which it occurs. A compound condition is considered covered if each simple condition it consists of is also covered. For example, $(x < 0 \texttt{ and } y < 0)$ is considered covered when both $x < 0$ and $y < 0$ have evaluated to `true` and `false` during one or more executions of the program in which they occur.

Decision coverage is concerned with the coverage of decisions regardless of whether or not a decision corresponds to a simple or a compound condition. Thus in the statement

```
1 if (x < 0 and y < 0)   2          z=foo(x,y);
```

there is only one decision that leads control to line 2 if the compound condition inside the `if` evaluates to `true`. However, a compound condition

might evaluate to `true` or `false` in one of several ways. For example, the condition at line 1 above evaluates to `false` when $x \geq 0$ regardless of the value of y. Another condition such as $x < 0$ `or` $y < 0$ evaluates to `true` regardless of the value of y when $x < 0$. With this evaluation characteristic in view, compilers often generate code that uses *short circuit* evaluation of compound conditions. For example, the `if` statement in the above code segment might get translated into the following sequence.

```
1   if (x<0)
2    if (y<0)  // Notice that y < 0 is evaluated only if x < 0 is true.
3      z=foo(x,y);
```

In the code segment above, we see two decisions, one corresponding to each simple condition in the `if` statement. This leads us to the following definition of condition coverage.

Condition coverage:

The condition coverage of T with respect to (P, R) is computed as $|C_c|/(|C_e| - |C_i|)$, where C_c is the set of simple conditions covered, C_i is the set of infeasible simple conditions, and C_e is the set of simple conditions in the program, i.e. the condition coverage domain. T is considered adequate with respect to the condition coverage criterion if the condition coverage of T with respect to (P, R) is 1.

The coverage domain in condition coverage is the set of all simple conditions in the program under test. Decision coverage implies condition coverage when there are no compound conditions.

Sometimes the following alternate formula is used to compute the condition coverage of a test:

$$\frac{|C_c|}{2 \times (|C_e| - |C_i|)}$$

where each simple condition contributes 2, 1, or 0 to C_c depending on whether it is covered, partially covered, or not covered, respectively. For example, when evaluating a test set T, if $x < y$ evaluates to `true` but never to `false`, then it is considered partially covered and contributes a 1 to C_c.

Example 7.14 Consider the following program that inputs values of x and y and computes the output z using functions `foo1` and `foo2`. Partial specifications for this program are given in Table 7.1. This table lists how z is computed for different combinations of x and y. A quick examination of P7.8 against Table 7.1 reveals that for $x \geq 0$ and $y \geq 0$ the program incorrectly computes z as `foo2`(x,y).

Table 7.1 Truth table for the computation of z in P7.8.

$x < 0$	$y < 0$	Output (z)
true	true	foo1(x,y)
true	false	foo2(x,y)
false	true	foo2(x,y)
false	false	foo1(x,y)

Program P7.8

```
begin
  int  x, y, z;
  input (x, y);
  if(x<0 and y<0)
    z= foo1(x,y);
  else
    z= foo2(x,y);
  output(z);
end
```

Consider T designed to test P7.8.

$T = \{t_1 :< x = -3, y = -2 >, t_2 :< x = -4, y = 2 >\}$

T is adequate with respect to the statement, block, and decision coverage criteria. You may verify that P7.8 behaves correctly on t_1 and t_2.

To compute the condition coverage of T, we note that $C_e = \{x < 0, y < 0\}$. Tests in T cover only the second of the two elements in C_e. As both conditions in C_e are feasible, $|C_i| = 0$. Plugging these values into the formula for condition coverage we obtain the condition coverage for T to be $1/(2-0) = 0.5$.

We now add the test $t_3 :< x = 3, y = 4 >$ to T. When executed against t_3, P7.8 incorrectly computes z as foo2(x,y). The output will be incorrect if `foo1(3, 4)` $\neq$ `foo2(3,4)`. The enhanced test set is adequate with respect to the condition coverage criterion and possibly reveals an error in the program.

7.2.5 Condition/decision coverage

In the previous two sections we learned that a test set is adequate with respect to decision coverage if it exercises all outcomes of each decision in the program during testing. However, when a decision is composed of a

compound condition, decision coverage does not imply that each simple condition within a compound condition has taken both values `true` and `false`.

Condition coverage ensures that each simple condition within a compound condition has assumed both values `true` and `false`. However, as illustrated in the next example, condition coverage does not require each decision to have produced both outcomes. Condition/decision coverage is also known as *branch condition coverage*.

Example 7.15 Consider a slightly different version of P7.8 obtained by replacing `and` by `or` in the `if` condition. For P7.9, we consider two test sets T_1 and T_2.

Program P7.9

```
begin
 int  x, y, z;
 input  (x,  y);
 if(x<0  or  y<0)
  z= foo1(x,y);
 else
  z= foo2(x,y);
 output(z);
end
```

$$T_1 = \left\{ \begin{array}{lll} t_1: & <x=-3, & y=2> \\ t_2: & <x=4, & y=2> \end{array} \right\}$$

$$T_2 = \left\{ \begin{array}{lll} t_1: & <x=-3, & y=2> \\ t_2: & <x=4, & y=-2> \end{array} \right\}$$

Test set T_1 is adequate with respect to the decision coverage criterion because test t_1 causes the `if` condition to be `true` and test t_2 causes it to be `false`. However, T_1 is not adequate with respect to the condition coverage criterion because condition $y < 0$ never evaluates to `true`. In contrast, T_2 is adequate with respect to the condition coverage criterion but not with respect to the decision coverage criterion.

The condition/decision coverage based adequacy criterion is developed to overcome the limitations of using the condition and decision coverage criteria independently. A definition follows.

Condition/decision coverage:

The condition/decision coverage of T with respect to (P, R) is computed as $(|C_c| + |D_c|)/((|C_e| - |C_i|) + (|D_e| - |D_i|))$, *where*

C_c denotes the set of simple conditions covered, D_c the set of decisions covered, C_e and D_e the sets of simple conditions and decisions, respectively, and C_i and D_i the sets of infeasible simple conditions and decisions, respectively. T is considered adequate with respect to the condition/decision coverage criterion if the condition/decision coverage of T with respect to (P, R) is 1.

Condition coverage does not imply decision coverage. This is true when one or more decisions in a program is made up of compound conditions.

Example 7.16 For P7.8, a simple modification of T_1 from Example 7.15 gives us T that is adequate with respect to the condition/decision coverage criteria.

$$T = \left\{ \begin{array}{lll} t_1: & < x = -3 & y = -2 > \\ t_2: & < x = 4 & y = 2 > \end{array} \right\}$$

7.2.6 Multiple condition coverage

Multiple condition coverage is also known as *branch condition combination coverage*. To understand multiple condition coverage, consider a compound condition that contains two or more simple conditions. Using condition coverage on some compound condition C implies that each simple condition within C has been evaluated to `true` and `false`. However, it does not imply that all combinations of the values of the individual simple conditions in C have been exercised. The next example illustrates this point.

Multiple condition coverage aims at covering combinations of values of the simple conditions in a compound condition. Of course, such combinations grow exponentially with the number of simple conditions in a compound condition.

Example 7.17 Consider $D = (A < B)$ `or` $(A > C)$ composed of two simple conditions $A < B$ and $A > C$. The four possible combinations of the outcomes of these two simple conditions are enumerated in Table 7.2.
Now consider test set T containing two tests.

$$T = \left\{ \begin{array}{llll} t_1: & < A = 2, & B = 3, & C = 1 > \\ t_2: & < A = 2, & B = 1, & C = 3 > \end{array} \right\}$$

Table 7.2 Combinations in $D = (A < B)$ `or` $(A > C)$.

	$A < B$	$A > C$	D
1	`true`	`true`	`true`
2	`true`	`false`	`true`
3	`false`	`true`	`true`
4	`false`	`false`	`false`

The two simple conditions in D are covered when evaluated against tests in T. However, only two combinations in Table 7.2, those at lines 1 and 4, are covered. We need two more tests to cover the remaining two combinations at lines 2 and 3 in Table 7.2. We modify T to T' by adding two tests that cover all combinations of values of the simple conditions in D.

$$T' = \left\{ \begin{array}{lllll} t_1: & <A=2 & B=3 & C=1> \\ t_2: & <A=2 & B=1 & C=3> \\ t_3: & <A=2 & B=3 & C=5> \\ t_4: & <A=2 & B=1 & C=1> \end{array} \right\}$$

To define test adequacy with respect to the multiple condition coverage criterion, suppose that the program under test contains a total of n decisions. Assume also that each decision contains $k_1, k_2, \ldots, k_n$ simple conditions. Each decision has several combinations of values of its constituent simple conditions. For example, decision i will have a total of 2^{k_i} combinations. Thus the total number of combinations to be covered is $(\Sigma_{i=1}^{n} 2^{k_i})$. With this background, we now define test adequacy with respect to multiple condition coverage.

Multiple condition coverage:

The multiple condition coverage of T with respect to (P, R) is computed as $|C_c|/(|C_e| - |C_i|)$, where C_c denotes the set of combinations covered, C_i the set of infeasible simple combinations, and $|C_e| = \Sigma_{i=1}^{n} 2^{k_i}$ the total number of combinations in the program. T is considered adequate with respect to the multiple condition coverage criterion if the multiple condition coverage of T with respect to (P, R) is 1.

The coverage domain for multiple condition coverage consists of all combinations of simple conditions in each compound condition in the program under test. Decisions made up of only simple conditions should also be included in this coverage domain.

Example 7.18 It is required to write a program that inputs values of integers A, B, and C and computes an output S as specified in Table 7.3. Note from this table the use of functions $f1$ through $f4$

Table 7.3 Computing S for P7.10.

	$A < B$	$A > C$	S
1	`true`	`true`	$f1(A, B, C)$
2	`true`	`false`	$f2(A, B, C)$
3	`false`	`true`	$f3(A, B, C)$
4	`false`	`false`	$f4(A, B, C)$

used to compute S for different combinations of the two conditions $A < B$ and $A > C$. P7.10 is written to meet the desired specifications. There is an obvious error in the program, computation of S for one of the four combinations, line 3 in the table, has been left out.

Program P7.10

```
1   begin
2     int  A, B, C, S=0;
3     input  (A, B, C);
4     if (A<B and A>C) S=f1(A, B, C);
5     if (A<B and A≤C) S=f2(A, B, C);
6     if (A≥B and A≤C) S=f4(A, B, C);
7      output(S);
8   end
```

Consider test set T developed to test P7.10; this is the same test used in Example 7.17.

$$T = \left\{ \begin{array}{lll} t_1: & <A=2, & B=3, \quad C=1> \\ t_2: & <A=2, & B=1, \quad C=3> \end{array} \right\}$$

P7.10 contains three decisions, six conditions, and a total of 12 combinations of simple conditions within the three decisions. Notice that because all three decisions use the same set of variables, A, B, and C, the number of distinct combinations is only four. Table 7.2 lists all four combinations.

When P7.10 is executed against tests in T, all simple conditions are covered. Also, the decisions at lines 4 and 6 are covered. However, the decision at line 5 is not covered. Thus, T is adequate with respect to condition coverage but not with respect to decision coverage. To improve the decision coverage of T, we obtain T' by adding test t_3 from Example 7.17.

$$T' = \left\{ \begin{array}{ll} t_1: & <A=2, B=3, C=1> \\ t_2: & <A=2, B=1, C=3> \\ t_3: & <A=2, B=3, C=5> \end{array} \right\}$$

T' is adequate with respect to decision coverage. However, none of the three tests in T' reveal the error in P7.10. Let us now evaluate whether or not T' is adequate with respect to the multiple condition coverage criteria.

Table 7.4 lists the 12 combinations of conditions in three decisions and the corresponding coverage with respect to tests in T'. From the table we find that one combination of conditions in each

Table 7.4 Condition coverage for P7.10.

	$A < B$	$A > C$	T	$A < B$	$A \leq C$	T^*	$A \geq B$	$A \leq C$	T
1	`true`	`true`	t_1	`true`	`true`	t_3	`true`	`true`	t_2
2	`true`	`false`	t_3	`true`	`false`	t_1	`true`	`false`	–
3	`false`	`true`	–	`false`	`true`	t_2	`false`	`true`	t_3
4	`false`	`false`	t_2	`false`	`false`	–	`false`	`false`	t_1

* T: Test

of the three decisions remains uncovered. For example, at line 3 the (`false`, `true`) combination of the two conditions $A < B$ and $A > C$ remains uncovered. To cover this pair we add t_4 to T' and get the following modified test set T''.

$$T'' = \left\{ \begin{array}{ll} t_1: & < A = 2, B = 3, C = 1 > \\ t_2: & < A = 2, B = 1, C = 3 > \\ t_3: & < A = 2, B = 3, C = 5 > \\ t_4: & < A = 2, B = 1, C = 1 > \end{array} \right\}$$

Test t_4 in T'' does cover all of the uncovered combinations in Table 7.4 and hence renders T'' adequate with respect to the multiple condition criterion.

You might have guessed that our analysis in Table 7.4 is redundant. As all three decisions in P7.10 use the same set of variables, A, B, and C, we need to analyze only one decision in order to obtain a test set that is adequate with respect to the multiple condition coverage criterion.

7.2.7 Linear code sequence and jump (LCSAJ) coverage

Test adequacy based on linear code sequence and jump aims at covering paths that might not otherwise be covered when decision, condition, or multiple condition coverage is used.

Execution of sequential programs that contain at least one condition proceeds in pairs where the first element of the pair is a sequence of statements (a block), executed one after the other, and terminated by a jump to the next such pair (another block). The first element of this pair is a sequence of statements that follow each other textually. The last such pair contains a jump to program exit thereby terminating program execution. An execution path through a sequential program is composed of one or more of such pairs.

A *Linear Code Sequence and Jump* is a program unit comprising a textual code sequence that terminates in a jump to the beginning of another code sequence and jump. The textual code sequence may contain one or more statements. An LCSAJ is represented as a triple (X, Y, Z) where X and Y are, respectively, locations of the first and the last statement and Z

is the location to which the statement at X jumps. The last statement in an LCSAJ is a jump and Z may be program exit.

When control arrives at statement X, follows through to statement Y, and then jumps to statement Z, we say that the LCSAJ (X, Y, Z) is *traversed*. Alternate terms for "traversed" are *covered* and *exercised*. The next three examples illustrate the derivation and traversal of LCSAJs in different program structures.

Example 7.19 Consider the following program consisting of one decision. We are not concerned with the code for function g used in this program.

Program P7.11

```
1   begin
2    int x, y, p;
3    input (x, y);
4    if(x<0)
5     p= g(y);
6    else
7     p= g(y*y);
8   end
```

Listed below are three LCSAJ's in P7.11. Note that each LCSAJ begins at a statement and ends in a jump to another LCSAJ. The jump at the end of an LCSAJ takes control to either another LCSAJ or to program exit.

LCSAJ	Start Line	End Line	Jump to
1	1	6	exit
2	1	4	7
3	7	8	exit

Now consider the following test set consisting of two test cases.

$$T = \left\{ \begin{array}{lll} t_1: & <\mathrm{x}=-5, & y=2> \\ t_2: & <\mathrm{x}=9, & y=2> \end{array} \right\}$$

When P7.11 is executed against t_1, LCSAJs (1, 6, exit) are excited. When the same program is executed again test t_2, the sequence of LCSAJs executed is (1, 4, 7) and (7, 8, exit). Thus, execution of P7.11 against both tests in T causes each of the three LCSAJs to be executed at least once.

Example 7.20 Consider the following program that contains a loop.

Program P7.12

```
1   begin
2   // Compute x^y given non-negative integers x and y.
3    int x, y, p;
4    input (x, y);
5    p=1;
6    count=y;
7    while(count>0){
8      p=p*x;
9      count=count-1;
10    }
11   output(p);
12   end
```

The LCSAJs in P7.12 are enumerated below. As before, each LCSAJ begins at a statement and ends in a jump to another LCSAJ.

LCSAJ	Start Line	End Line	Jump to
1	1	10	7
2	7	10	7
3	7	7	11
4	1	7	11
5	11	12	exit

The following test set consisting of three test cases traverses each of the five LCSAJs listed above.

$$T = \left\{ \begin{array}{lll} t_1: & <x=5 & y=0> \\ t_2: & <x=5 & y=2> \end{array} \right\}$$

Upon execution on t_1, P7.12 traverses LCSAJ (1, 7, 11) followed by LCSAJ (11, 12, exit). When P7.12 is executed against t_2, the LCSAJs are executed in the following sequence: (1, 10, 7) → (7, 10, 7) → (7, 7, 11) → (11, 12, exit).

Example 7.21 Consider the following program that contains several conditions.

Program P7.13

```
1   begin
2     int x, y, p;
```

```
3     input (x, y);
4     p= g(x);
5     if(x<0)
6        p= g(y);
7     if(p<0)
8       q= g(x);
9     else
10      q= g(x*y);
11   end
```

Five LCSAJs for P7.13 follow.

LCSAJ	Start Line	End Line	Jump to
1	1	9	exit
2	1	5	7
3	7	10	exit
4	1	7	10
5	10	10	exit

The following test set consisting of two test cases traverses each of the five LCSAJs listed above.

$$T = \left\{ \begin{array}{lll} t_1: & <\mathrm{x}=-5 & y=0> \\ t_2: & <\mathrm{x}=5 & y=2> \\ t_3: & <\mathrm{x}=-5 & y=2> \end{array} \right\}$$

Assuming that g(0)<0, LCSAJ 1 is traversed when P7.13 is executed against t_1. In this case, the decisions at lines 5 and 7 both evaluate to true. Assuming that g(5)≥0, the sequence of LCSAJs executed when P7.13 is executed against t_2 is (1, 5, 7) → (7, 10, exit). Both decisions evaluate to false during this execution.

We note that the execution of P7.13 against t_1 and t_2 has covered both the decisions. Hence, T is adequate with respect to the decision coverage criterion even if test t_3 were not included. However, LCSAJs 4 and 5 have not been traversed yet. Assuming that g(2)<0, the remaining two LCSAJs are traversed, LCSAJ 4 followed by LCSAJ 5, when P7.13 is executed against t_3.

Example 7.21 illustrates that a test adequate with respect to decision coverage might not exercise all LCSAJs and more than one decision statement might be included in one LCSAJ, as for example is the case with LCSAJ 1. We now give a formal definition of test adequacy based on LCSAJ coverage.

LCSAJ coverage:

The LCSAJ coverage of a test set T with respect to (P, R) is computed as

$$\frac{\text{Number of LCSAJs exercised}}{\text{Total number of feasible LCSAJs}}$$

T is considered adequate with respect to the LCSAJ coverage criterion if the LCSAJ coverage of T with respect to (P, R) is 1.

Test adequacy based on LCSAJ's is computed, as in other cases, as the ratio of the number of LCSAJs covered to the number of feasible LCSAJs. Note that there are multiple ways to define an LCSAJ depending on how long a sequence of code sequence and jump is one willing to consider.

7.2.8 Modified condition/decision coverage

As we learned in the previous section, multiple condition coverage requires covering *all* combinations of simple conditions within a compound condition. Obtaining multiple condition coverage might become expensive when there are many embedded simple conditions.

When a compound condition C contains n simple conditions, the maximum number of tests required to cover C is 2^n. Table 7.5 exhibits the growth in the number of tests as n increases. The table also shows the time it will take to run all tests given that it takes 1 millisecond (ms) to set up and execute each test. It is evident from the numbers in this table that it would be impractical to obtain 100% the multiple condition coverage for complex conditions. One might ask: "Would any programmer ever devise a complex condition that contains 32 simple conditions?" Indeed, though not frequently, some avionics systems do contain such complex conditions.

The Modified condition/decision coverage is also known as MC/DC coverage. In several application domains, e.g., aerospace, tests must be demonstrated to be MC/DC adequate.

A weaker adequacy criterion based on the notion of modified condition decision coverage, also known as "MC/DC coverage," allows a thorough yet practical test of all conditions and decisions. As is implied in its name, there are two parts to this criteria: the "MC" part and the "DC" part. The DC part corresponds to decision coverage discussed earlier.

Table 7.5 Growth in the maximum number of tests required for multiple condition coverage for a condition with n simple conditions.

n	Number of tests	Time to execute all tests
1	2	2 ms
4	16	16 ms
8	256	256 ms
16	65536	65.5 seconds
32	4294967296	49.5 days

The next example illustrates the meaning of the "MC" part in the MC/DC coverage criteria. Be forewarned that this example is merely illustrative of the meaning of "MC" and is not to be treated as an illustration of how one could obtain MC/DC coverage. A practical method for obtaining MC/DC coverage is given later in this section.

Example 7.22 To understand the "MC" portion of the MC/DC coverage, consider a compound condition $C = (C_1$ `and` $C_2)$ `or` C_3 where C_1, C_2, and C_3 are simple conditions. To obtain MC adequacy, we must generate tests to show that each simple condition affects the outcome of C *independently*. To construct such tests, we fix two of the three simple conditions in C and vary the third. For example, we fix C_1 and C_2 and vary C_3 as shown in the first eight rows of Table 7.6. There are three combinations of two simple conditions and each combination leads to eight possibilities. Thus we have a total of 24 rows in Table 7.6.

Many of these 24 rows are identical as indicated in the table. For each of the three simple conditions, we select one set of two tests that demonstrate the independent effect of that condition on C. Thus we select tests (3, 4) for C_3, (11, 12) for C_2, and (19, 20) for C_1. These tests are shown in Table 7.7 which contains a total of only six tests. Note that we could have as well selected (5, 6) or (7, 8) for C_3.

Table 7.7 also has some redundancy as tests (2, 4) and (3, 5) are identical. Compacting this table further by selecting only one amongst several identical tests, we obtain a minimal MC adequate test set for C shown in Table 7.8.

The key idea in Example 7.22 is that every compound condition in a program must be tested by demonstrating that each simple condition within the compound condition has an independent effect on its outcome. The example also reveals that such demonstration leads to fewer tests than required by the multiple-condition coverage criteria. For example, a total of eight tests are required to satisfy the multiple condition criteria when condition $C = (C_1$ `and` $C_2)$ `or` C_3 is tested. This is in contrast to only four tests required to satisfy the MC/DC criterion.

MC part of the MC/DC adequacy is demonstrated by showing that each simple condition in each decision in the program under test has an impact on the outcome of the decision. Doing so generally requires fewer tests than would be required to obtain multiple condition coverage.

7.2.9 MC/DC adequate tests for compound conditions

It is easy to improve upon the brute force method of Example 7.22 for generating MC/DC adequate tests for a condition. First, we note that only two tests are required for a simple condition. For example, to cover a simple condition $x < y$, where x and y are integers, one needs only two

Table 7.6 Test cases for $C = (C_1$ `and` $C_2)$ `or` C_3 to illustrate MC/DC coverage. Identical rows are listed in parentheses.

Test	Inputs			Output
	C_1	C_2	C_3	C
Fix C_1 and C_2 to `true`, vary C_3.				
1(9, 17)	`true`	`true`	`true`	`true`
2(11, 19)	`true`	`true`	`false`	`true`
* **Fix C_1 to `true`, C_2 to `false`, vary C_3.**				
3(10, 21)	`true`	`false`	`true`	`true`
4(12, 23)	`true`	`false`	`false`	`false`
* **Fix C_1 to `false`, C_2 to `true`, vary C_3.**				
5(13, 18)	`false`	`true`	`true`	`true`
6(15, 20)	`false`	`true`	`false`	`false`
* **Fix C_1 and C_2 to `false`, vary C_3.**				
7(14, 22)	`false`	`false`	`true`	`true`
8(16, 24)	`false`	`false`	`false`	`false`
Fix C_1 and C_3 to `true`, vary C_2.				
9(1, 17)	`true`	`true`	`true`	`true`
10(3, 21)	`true`	`false`	`true`	`true`
* **Fix C_1 to `true`, C_3 to `false`, vary C_2.**				
11(2, 19)	`true`	`true`	`false`	`true`
12(4, 23)	`true`	`false`	`false`	`false`
Fix C_1 to `false`, C_3 to `true`, vary C_2.				
13(5, 18)	`false`	`true`	`true`	`true`
14(7, 22)	`false`	`false`	`true`	`true`
Fix C_1 and C_3 to `false`, vary C_2.				
15(6, 20)	`false`	`true`	`false`	`true`
16(8, 24)	`false`	`false`	`false`	`true`
Fix C_2 and C_3 to `true`, vary C_1.				
17(1, 9)	`true`	`true`	`true`	`true`
18(5, 13)	`false`	`true`	`true`	`true`
* **Fix C_2 to `true`, C_3 to `false`, vary C_1.**				
19(2, 11)	`true`	`true`	`false`	`true`
20(6, 15)	`false`	`true`	`false`	`false`
Fix C_2 to `false`, C_3 to `true`, vary C_1.				
21(3, 10)	`true`	`false`	`true`	`true`
22(7, 14)	`false`	`false`	`true`	`true`
Fix C_2 to `false`, C_3 to `false`, vary C_1.				
23(4, 12)	`true`	`false`	`false`	`false`
24(8, 16)	`false`	`false`	`false`	`false`

*Corresponding tests affect C and may be included in the MC/DC adequate test set.

Table 7.7 MC-adequate tests for $C = (C_1$ `and` $C_2)$ `or` C_3

Test	C_1	C_2	C_3	C	Effect demonstrated for
1[3]	`true`	`false`	`true`	`true`	
2[4]	`true`	`false`	`false`	`false`	C_3
3[11]	`true`	`true`	`false`	`true`	
4[12]	`true`	`false`	`false`	`false`	C_2
5[19]	`true`	`true`	`false`	`true`	
6[20]	`false`	`true`	`false`	`false`	C_1

[x]: x is the corresponding test in Table 7.6.

Table 7.8 Minimal MC-adequate tests for $C = (C_1$ `and` $C_2)$ `or` C_3

Test	C_1	C_2	C_3	C	Comments
t_1	`true`	`false`	`true`	`true`	Tests t_1 and t_2 cover C_3.
t_2	`true`	`false`	`false`	`false`	Tests t_2 and t_3 cover C_2.
t_3	`true`	`true`	`false`	`true`	Tests t_3 and t_4 cover C_1.
t_4	`false`	`true`	`false`	`false`	

tests, one that causes the condition to be `true` and another that causes it to be `false`.

Next, determine MC/DC adequate tests for compound conditions that contain two simple conditions. Table 7.9 lists adequate tests for such compound conditions. Note that three tests are required to cover each condition using the MC/DC requirement. This number would be four if multiple condition coverage is required. It is instructive to carefully go through each of the three conditions listed in Table 7.9 and verify that indeed the tests given are independent (also try Exercise 7.14).

Fewer tests are needed constructed to satisfy the MC/DC coverage criterion than for the multiple condition coverage criterion.

We now build Table 7.10, that is analogous to Table 7.9, for compound conditions that contain three simple conditions. Notice that only four tests are required for to cover each compound condition listed in Table 7.10. One can generate the entries in Table 7.10 from Table 7.9 by using the following procedure which works for a compound condition C that is a conjunct of three simple conditions, i.e. $C = (C_1$ `and` C_2 `and` $C_3)$.

There is a simple procedure for finding combinations of truth values of simple conditions that satisfy MC/DC criteria. Of course, actual tests need to be derived from these truth values.

1. Create a table with four columns and four rows. Label the columns as **Test**, C_1, C_2, C_3, and C, from left to right. The column labeled **Test** contains rows labeled by test case numbers t_1 through t_4. The remaining entries are empty. The last column labeled **Comments** is optional.

Table 7.9 MC/DC adequate tests for compound conditions that contain two simple conditions.

Test	C_1	C_2	C	Comments
Condition: $C_a = (C_1$ `and` $C_2)$				
t_1	`true`	`true`	`true`	Tests t_1 and t_2 cover C_2.
t_2	`true`	`false`	`false`	
t_3	`false`	`true`	`false`	Tests t_1 and t_2 cover C_1.
MC/DC adequate test set for $C_a = \{t_1, t_2, t_3\}$				

Condition: $C_b = (C_1$ `or` $C_2)$				
t_4	`false`	`true`	`true`	Tests t_4 and t_5 cover C_2.
t_5	`false`	`false`	`false`	
t_6	`true`	`false`	`true`	Tests t_5 and t_6 cover C_1.
MC/DC adequate test set for $C_b = \{t_4, t_5, t_6\}$				

Condition: $C_c = (C_1$ `xor` $C_2)$				
t_7	`true`	`true`	`false`	Tests t_7 and t_8 cover C_2.
t_8	`true`	`false`	`true`	
t_9	`false`	`false`	`false`	Tests t_8 and t_9 cover C_1.
MC/DC adequate test set for $C_c = \{t_7, t_8, t_9\}$				

Test	C_1	C_2	C_3	C	Comments
t_1					
t_2					
t_3					
t_4					

2. Copy all entries in columns C_1, C_2, and C from Table 7.9 into columns C_2, C_3, and C of the empty table.

Test	C_1	C_2	C_3	C	Comments
t_1		`true`	`true`	`true`	
t_2		`true`	`false`	`false`	
t_3		`false`	`true`	`false`	
t_4					

3. Fill the first three rows in the column marked C_1 with `true` and the last row with `false`.

Test	C_1	C_2	C_3	C	Comments
t_1	`true`	`true`	`true`	`true`	
t_2	`true`	`true`	`false`	`false`	
t_3	`true`	`false`	`true`	`false`	
t_4	`false`				

4. Fill entries in the last row under columns labeled C_2, C_3, and C with `true`, `true`, and `false`, respectively.

Test	C_1	C_2	C_3	C	Comments
t_1	`true`	`true`	`true`	`true`	Tests t_1 and t_2 cover C_3.
t_2	`true`	`true`	`false`	`false`	
t_3	`true`	`false`	`true`	`false`	Tests t_1 and t_3 cover C_2.
t_4	`false`	`true`	`true`	`false`	Tests t_1 and t_4 cover C_1.

5. We now have a table containing MC/DC adequate tests for $C = (C_1$ `and` C_2 `and` $C_3)$ derived from tests for $C = (C_1$ `and` $C_2)$.

The procedure illustrated above can be extended to derive tests for any compound condition using tests for a simpler compound condition (see Exercises 7.15 and 7.16). The important point to note here is that for any compound condition, the size of an MC/DC adequate test set grows *linearly* in the number of simple conditions. Table 7.5 is reproduced below with columns added to compare the minimum number of tests required, and the time to execute them, for multiple condition and MC/DC coverage criteria.

7.2.10 Definition of MC/DC coverage

An MC/DC adequate test set covers all decisions, all simple conditions, and has demonstrated the effect of each simple condition on the outcome of the decision in which the simple condition is embedded.

We now provide a more complete definition of the MC/DC coverage. A test set T for program P written to meet requirements R is considered adequate with respect to the MC/DC coverage criterion if upon the execution of P on each test in T, the following requirements are met.

1. Each block in P has been covered.
2. Each simple condition in P has taken both `true` and `false` values.

Table 7.10 MC/DC adequate tests for compound conditions that contain three simple conditions.

Test	C_1	C_2	C_3	C	Comments
Condition: $C_a = (C_1$ `and` C_2 `and` $C_3)$					
t_1	`true`	`true`	`true`	`true`	Tests t_1 and t_2 cover C_3.
t_2	`true`	`true`	`false`	`false`	
t_3	`true`	`false`	`true`	`false`	Tests t_1 and t_3 cover C_2.
t_4	`false`	`true`	`true`	`false`	Tests t_1 and t_2 cover C_1.
MC/DC adequate test set for $C_a = \{t_1, t_2, t_3, t_4\}$					
Condition: $C_b = (C_1$ `or` C_2 `or` $C_3)$					
t_5	`false`	`false`	`false`	`false`	Tests t_5 and t_6 cover C_3.
t_6	`false`	`false`	`true`	`true`	
t_7	`false`	`true`	`false`	`true`	Tests t_5 and t_7 cover C_2.
t_8	`true`	`false`	`false`	`true`	Tests t_5 and t_8 cover C_1.
MC/DC adequate test set for $C_b = \{t_5, t_6, t_7, t_8\}$					
Condition: $C_c = (C_1$ `xor` C_2 `xor` $C_3)$					
t_9	`true`	`true`	`true`	`true`	Tests t_9 and t_{10} cover C_3.
t_{10}	`true`	`true`	`false`	`false`	
t_{11}	`true`	`false`	`true`	`false`	Tests t_9 and t_{11} cover C_2.
t_{12}	`false`	`true`	`true`	`false`	Tests t_9 and t_{12} cover C_1.
MC/DC adequate test set for $C_c = \{t_9, t_{10}, t_{11}, t_{12}\}$					

3. Each decision in P has taken all possible outcomes.
4. Each simple condition within a compound condition C in P has been shown to independently effect the outcome of C. *This is the MC part of the coverage we discussed in detail earlier in this section.*

The first three requirements above correspond to block, condition, and decision coverage, respectively, and have been discussed in earlier sections. The fourth requirement corresponds to "MC" coverage discussed earlier in this section. Thus the MC/DC coverage criterion is a mix of four coverage criteria based on the flow of control. With regard to the second requirement, it is to be noted that conditions that are not part of a decision, such as the one in the following statement

$A = (p < q)$ `or` $(x > y)$

are also included in the set of conditions to be covered. With regard to the fourth requirement, a condition such as (A `and` B) `or` (C `and` A) poses

Table 7.11 MC/DC adequacy and the growth in the least number of tests required for a condition with n simple conditions.

n	Minimum tests		Time to execute all tests	
	Multiple condition	MC/DC	Multiple condition	MC/DC
1	2	2	2 ms	2 ms
4	16	5	16 ms	5 ms
8	256	9	256 ms	9 ms
16	65536	17	65.5 seconds	17 ms
32	4294967296	33	49.5 days	33 ms

a problem. It is not possible to keep the first occurrence of A fixed while varying the value of its second occurrence. Here the first occurrence of A is said to be *coupled* to its second occurrence. In such cases, an adequate test set need only demonstrate the independent effect of any one occurrence of the coupled condition.

There are multiple ways of quantitatively measuring MC/DC adequacy. One way is to treat separately each of the individual requirements in the MC/DC criterion. The other way is to create a single metric.

A numerical value can also be associated with a test to determine the extent of its adequacy with respect to the MC/DC criterion. One way to do so is to treat separately each of the four requirements listed above for MC/DC adequacy. Thus, four distinct coverage values can be associated with T, namely, block coverage, condition coverage, decision coverage, and MC coverage. The first three of these four have already been defined earlier. A definition of MC coverage follows.

Let $C_1, C_2, \ldots, C_N$ be the conditions in P; each condition could be simple or compound and may or may not appear within the context of a decision. Let n_i denote the number of simple conditions in C_i, e_i the number of simple conditions shown to have independent affect on the outcome of C_i, and f_i the number of infeasible simple conditions in C_i. Note that a simple condition within a compound condition C is considered infeasible if it is impossible to show its independent affect on C while holding constant the remaining simple conditions in C. The MC coverage of T for program P subject to requirements R, denoted by MC_c, is computed as follows.

$$\mathrm{MC}_c = \frac{\Sigma_{i=1}^{N} e_i}{\Sigma_{i=1}^{N} (n_i - f_i)}$$

Thus, test set T is considered adequate with respect to the MC coverage if MC_c of T is 1. Having now defined all components of the MC/DC coverage, we are ready for a complete definition of the adequacy criterion based on this coverage.

Modified condition/decision coverage:

A test T to test program P subject to requirements R is considered adequate with respect to the MC/DC coverage criterion if T is adequate with respect to block, decision, condition, and MC coverage.

The next example illustrates the process of generating MC/DC adequate tests for a program that contains three decisions, one composed of a simple condition and the remaining two composed of compound conditions.

Example 7.23 P7.14 is written to meet the following requirements.

R_1: Given coordinate positions `x`, `y`, and `z`, and a direction value *d*, the program must invoke one of the three functions `fire-1`, `fire-2`, and `fire-3` as per conditions below.

$R_{1.1}$: Invoke `fire-1` when (`x<y`) `and` (`z * z > y`) `and` (prev="East"), where *prev* and *current* denote, respectively, the previous and current values of *d*.

$R_{1.2}$: Invoke `fire-2` when (`x<y`) `and` (`z * z ≤ y`) `or` (current="South").

$R_{1.3}$: Invoke `fire-3` when none of the two conditions above is `true`.

R_2: The invocation described above must continue until an input Boolean variable becomes `true`.

Program P7.14

```
begin
float  x, y, z;
direction  d;
string  prev, current;
bool  done;
input (done);
current="North";
while  (¬ done) {      ← Condition C1.
input (d);
        prev=current;  current=f(d);
        input (x, y, z);
        if ((x<y) and (z * z > y) and (prev=="East"))
    ←   Condition C2.
     fire-1(x, y);
     else if ((x<y and (z * z ≤ y) or (current==''South''))
     ←   Condition C3.
```

```
15          fire-2(x, y);
16        else
17          fire-3(x, y);
18        input (done);
19        }
20        output("Firing completed.") ;
21    end
```

First we generate tests to meet the given requirements. Three tests derived to test $R_{1.1}$, $R_{1.2}$, and $R_{1.3}$ follow; note that these tests are to be executed in the sequence listed.

Test set T_1 for P7.14						
Test	**Requirement**	done	d	x	y	z
t_1	$R_{1.2}$	false	East	10	15	3
t_2	$R_{1.1}$	false	South	10	15	4
t_3	$R_{1.3}$	false	North	10	15	5
t_4	R_2	true	–	–	–	–

Assuming that P7.14 has been executed against the tests given above, let us analyze which decisions have been covered and which ones have not been covered. To make this task easy, the values of all simple and compound conditions are listed below for each test case.

Test	C_1	**Comment**
t_1	true	C_2=false, C_3=true, hence fire-2 invoked.
t_2	true	C_2=true, hence fire-1 invoked.
t_3	true	Both C_2 and C_3 are false, hence fire-3 invoked.
t_4	false	This terminates the loop. The decision at line 8 is covered.

First, it is easy to verify that all statements in P7.14 are covered by test set T_1. This also implies that all blocks in this program are covered. Next, a quick examination of the tables above reveals that the three decisions at lines 8, 12, and 14 are also covered as both outcomes of each decision have been taken. Condition C_1 is covered by tests t_1 and t_4 and also by tests t_2 and t_4, and by t_3 and t_4. However, compound condition C_2 is not covered because $x < y$ is not covered. Also, C_3 is not covered because $(x<y)$ is not covered. This analysis leads to the conclusion that T_1 is adequate with respect to the statement, block, and decision coverage criteria but not with respect to the condition coverage criteria.

Next we modify T_1 to make it adequate with respect to the condition coverage criteria. Recall that C_2 is not covered because $x < y$ is not covered.

C_2=($x<y$) and ($z * z > y$) and (prev=="East")					
Test	$x<y$	$z * z > y$	prev=="East"	C_2	**Comment**
t_1	true	false	false	false	
t_2	true	true	true	true	fire-1 invoked. Decision at line 12 is covered by t_1 and t_2 and also by t_2 and t_3.
t_3	true	true	false	false	
t_4	–	–	–	–	Condition not evaluated, loop terminates.

C_3=($x<y$) and ($z * z \leq y$) or (current=="South")					
Test	$x<y$	$z * z \leq y$	current== "South"	C_3	**Comment**
t_1	true	true	false	true	fire-2 invoked.
t_2	–	–	–	–	Condition not evaluated.
t_3	true	false	false	false	Decision at line 14 is covered by t_1 and t_3.
t_4	–	–	–	–	Condition not evaluated, loop terminates.

To cover $x < y$ we generate a new test t_5 and add it to T_1 to obtain T_2 given below. The value of x being greater than that of y which makes $x < y$ false. Also, we looked ahead and set d to "South" to make sure that (current == "South") in C_3 evaluates to true. Note the placement of t_5 with respect to t_4 to ensure that the program under test is executed against t_5 *before* it is executed against t_4 or else the program will terminate without covering $x < y$.

Test set T_2 for P7.14						
Test	**Requirement**	done	d	x	y	z
t_1	$R_{1.2}$	false	East	10	15	3
t_2	$R_{1.1}$	false	South	10	15	4
t_3	$R_{1.3}$	false	North	10	15	5
t_5	$R_{1.1}$ and $R_{1.2}$	false	South	10	5	5
t_4	R_2	true	–	–	–	–

The evaluations of C_2 and C_3 are reproduced below with a row added for the new test t_5. Note that test T_2, obtained by enhancing T_1, is adequate with respect to the condition coverage criterion. Obviously, it is not adequate with respect to the multiple condition coverage criteria. Such adequacy will imply a minimum of eight tests and the development of these is left to Exercise 7.21.

C_2=($x<y$) and ($z * z > y$) and (prev==“East”)					
Test	$x<y$	$z * z > y$	prev== “East”	C_2	**Comment**
t_1	true	false	false	false	
t_2	true	true	true	true	fire-1 invoked.
t_3	true	true	false	false	
t_5	false	true	false	false	C_2 is covered because each of its component conditions is covered.
t_4	–	–	–	–	Condition not evaluated, loop terminates.

C_3=($x<y$) and ($z * z \leq y$) or (current==“South”)					
Test	$x<y$	$z * z \leq y$	current== “South”	C_3	**Comment**
t_1	true	true	false	true	fire-2 invoked.
t_2	–	–	–	–	Condition not evaluated.
t_3	true	false	false	false	
t_5	false	false	true	true	C_3 is covered because each of its component conditions is covered.
t_4	–	–	–	–	Condition not evaluated, loop terminates.

Next, we check if T_2 is adequate with respect to the MC/DC criterion. From the table given above for C_2, we note that conditions ($x<y$) and ($z*z>y$) are kept constant in t_2 and t_3, while (prev==“East”) is varied. These two tests demonstrate the independent effect of (prev==“East”) on C_2. However, the independent effect of the remaining two conditions

is not demonstrated by T_2. In the case of C_3, we note that in tests t_1 and t_3, conditions (`x<y`) and (current==“South”) are held constant while (`z* z≤y`) is varied. These two tests demonstrate the independent effect of (`z* z≤y`) on C_3. Tests t_1 and t_3 demonstrate the independent effect of (`z* z ≤y`) on C_3. However, the independent effect of (current==“South”) on C_3 is not demonstrated by T_2. This analysis reveals that we need to add at least two tests to T_2 to obtain MC/DC coverage.

To obtain MC/DC coverage for C_2, consider (`x<y`). We need two tests that fix the remaining two conditions, vary (`x<y`), and cause C_2 to be evaluated to `true` and `false`. We reuse t_2 as one of these two tests. The new test must hold (`z* z>y`) and (prev==“East”) to `true` and make (`x<y`) evaluate to `false`. One such test is t_6, to be executed right after t_1 but before t_2. Execution of the program against t_6 causes (`x<y`) to evaluate to `false` and C_2 also to `false`. t_6 and t_2 together demonstrate the independent effect of (`x<y`) on C_2. Using similar arguments, we add test t_7 to show the independent effect of (`z* z>y`) on C_2. Adding t_6 and t_7 to T_2 gives us an enhanced test set T_3.

Test set T_3 for P7.14						
Test	**Requirement**	`done`	`d`	`x`	`y`	`z`
t_1	$R_{1.2}$	`false`	East	10	15	3
t_6	R_1	`false`	East	10	5	3
t_7	R_1	`false`	East	10	15	3
t_2	$R_{1.1}$	`false`	South	10	15	4
t_3	$R_{1.3}$	`false`	North	10	15	5
t_5	$R_{1.1}$ and $R_{1.2}$	`false`	South	10	5	5
t_8	$R_{1.2}$	`false`	South	10	5	2
t_9	$R_{1.2}$	`false`	North	10	5	2
t_4	R_2	`true`	–	–	–	–

Upon evaluating C_3 against the newly added tests, we observe that the independent effects of (`x<y`) and (`z* z ≤ y`) have been demonstrated. We add t_8 and t_9 to show the independent effect of (current==“South”) on C_3. The complete test set, T_3 listed above, is MC/DC adequate for P7.14. Once again, note the importance of test sequencing. t_1 and t_7 contain identical values of input variables but lead to different effects in the program due to their position in the sequence in which tests are executed. Also note that test t_2 is of no use for covering any portion of C_3 because C_3 is not evaluated when the program is executed against this test.

C_2=(x<y) and (z * z > y) and (prev==“East”)					
Test	x<y	$z * z > y$	prev== “East”	C_2	**Comment**
t_1	true	false	false	false	
t_6	false	true	true	false	
t_7	true	false	true	false	t_7 and t_2 demonstrate the independent effect of $z * z > y$ on C_2.
t_2	true	true	true	true	t_6 and t_2 demonstrate the independent effect of $x < y$ on C_2.
t_3	true	true	false	false	t_2 and t_3 demonstrate the independent effect of (prev==“East”) on C_2.
t_5	false	true	false	false	
t_8	false	false	false	false	
t_9	false	false	true	false	
t_4	–	–	–	–	Conditions not evaluated.

C_3=(x<y) and ($z * z \leq y$) or (current=“South”)					
Test	x<y	$z * z \leq y$	current== “South”	C_3	**Comment**
t_1	true	true	false	true	fire-2 invoked.
t_6	false	false	false	false	
t_7	true	true	false	true	t_7 and t_9 demonstrate the independent effect of ($x < y$) on C_3
t_2	–	–	–	–	Conditions not evaluated.
t_3	true	false	false	false	t_1 and t_3 demonstrate the independent effect of $z * z \leq y$ on C_3.
t_5	false	false	true	true	
t_8	false	true	true	true	
t_9	false	true	false	false	t_8 and t_9 demonstrate the independent effect of (current == “South”) on C_3.
t_4	–	–	–	–	Conditions not evaluated.

7.2.11 Minimal MC/DC tests

MC/DC tests generated using techniques described earlier might not be minimal. Thus, it may be possible to discard some of these tests without affecting the test adequacy. Such reduction can be achieved using techniques discussed elsewhere in this book.

In Example 7.23, we did not make any effort to obtain the smallest test set adequate with respect to the MC/DC coverage. However, while generating tests for large programs involving several compound conditions, one might like to generate the smallest number of MC/DC adequate tests. This is because execution against each test case might take a significant chunk of test time, e.g. 24 hours! Such execution times might be considered exorbitant in several development environments. In situations like this, one might consider using one of several techniques for the minimization of test sets. Such techniques are discussed in Chapter 9.

7.2.12 Error detection and MC/DC adequacy

In Example 7.14, we saw how obtaining condition coverage led to a test case that revealed a missing condition error. Let us now dwell on what types of errors might one find while enhancing tests using the MC/DC criterion. In this discussion we assume that the test being enhanced is adequate with respect to condition coverage but not with respect to MC/DC coverage, and has not revealed an error. Consider the following three error types in a compound condition; errors in simple conditions have been discussed in Section 7.2.3.

Missing condition: One or more simple conditions is missing from a compound condition. For example, the correct condition should be ($x < y$ `and` done) but the condition coded is (done).

Incorrect Boolean operator: One or more Boolean operators is incorrect. For example, the correct condition is ($x < y$ `and` done) which has been coded as ($x < y$ `or` done).

Mixed type: One or more simple conditions is missing and one or more Boolean operators is incorrect. For example, the correct condition should be ($x < y$ `and` $z * x \geq y$ `and` d="South") has been coded as ($x < y$ `or` $z * x \geq y$).

Note that we are considering errors in code. We assume that we are given a test set T for program P subject to requirements R. T is adequate with respect to the condition coverage criterion. The test team has decided to enhance T to make it adequate with respect to the MC/DC coverage criterion. Obviously the enhancement process will lead to the addition of zero or more tests to T; no tests will be added if T is already MC/DC adequate.

We also assume that P contains a missing condition error or an incorrect Boolean operator error. The question we ask is: "Will the enhanced

T detect the error?" In the three examples that follow, we do not consider the entire program, instead, we focus only on the erroneous condition (see Exercise 7.25).

Example 7.24 Suppose that condition $C = C_1$ `and` C_2 `and` C_3 has been coded in a program as $C' = C_1$ `and` C_3. The first two tests below form a test set adequate with respect to condition coverage for C'. The remaining two tests, when combined with the first two, constitute an MC/DC adequate test for C'. Note that the adequate test set is developed for the condition as coded in the program, and not the correct condition.

	Test	C	C'	Error detected
	C_1, C_2, C_3	C_1 and C_2 and C_3	C_1 and C_3	
t	true, true, true	true	true	No
t	false, false, false	false	false	No
t	true, true, false	false	false	No
t	false, false, true	false	false	No

There is no guarantee that an MC/DC adequate test set will be able to detect a missing-condition error.

The first two tests do not reveal the error as both C and C' evaluate to the same value for each test. The next two tests, added to obtain MC/DC coverage, also do not detect the error. Thus, in this example, an MC/DC adequate test is unable to detect a missing-condition error.

It should be noted in the above example that the first two tests are not adequate with respect to the condition coverage criterion if short-circuit evaluation is used for compound conditions. Of course, this observation does not change the conclusion we arrived at. This conclusion can be generalized for a conjunction of n simple conditions (see Exercise 7.24).

Test adequacy is affected by short-circuit evaluation of conditions. For example, a test set might be adequate with respect to the condition coverage criterion in the absence of short-circuit evaluation but not otherwise.

Example 7.25 Suppose that condition $C = C_1$ `and` C_2 `and` C_3 has been coded as $C' = (C_1$ `or` $C_2)$ `and` C_3. Six tests that form an MC/DC adequate set are in the following table. Indeed, this set does reveal the error as $t_{7.12}$ and $t_{7.12}$ cause C and C' to evaluate differently. However, the first two tests do not reveal the error. Once again, we note that the first two tests below are not adequate with respect to the condition coverage criterion if short-circuit evaluation is used for compound conditions.

	Test	C	C'	Error detected
	C_1, C_2, C_3	C_1 and C_2 and C_3	(C_1 or C_2) and C_3	
t	true, true, true	true	true	No
t	false, false, false	false	false	No
t	true, true, false	false	false	No
t	true, false, true	false	true	Yes
t	false, true, true	false	true	Yes
t	false, false, true	false	false	No

Example 7.26 Suppose that condition $C = C_1$ or C_2 or C_3 has been coded as $C = C_1$ and C_3. Four MC/DC adequate tests are given below. All but $t_{7.12}$ reveal the error.

	Test	C	C'	Error detected
	C_1, C_2, C_3	$C = C_1$ or C_2 or C_3	C_1 and C_3	
t	true, true, true	true	true	No
t	false, true, false	true	false	Yes
t	true, true, false	true	false	Yes
t	false, true, true	true	false	Yes

It can be shown that MC/DC adequate tests have superior error detection ability than those adequate only with respect to the condition coverage criterion.

Not surprisingly, examples given above show that satisfying the MC/DC adequacy criteria does not necessarily imply that errors made while coding conditions will be revealed. However, the examples do favor MC/DC over condition coverage. The first two tests in each of the three examples above are also adequate with respect to decision coverage. Hence these examples also show that an MC/DC adequate test will *likely* reveal more errors than a decision or condition-coverage adequate test. Note the emphasis on the word "likely." MC/DC adequacy does not guarantee the detection of errors in an incorrectly coded condition.

7.2.13 Short-circuit evaluation and infeasibility

Short-circuit evaluation refers to the method of partially evaluating a compound condition whenever it is sufficient to do so. It is also known as *lazy evaluation*. C requires short-circuit evaluation and Java allows both.

For example, consider the following decision comprising a conjunction of conditions: (C_1 `and` C_2).

Short-circuit evaluation of predicates might lead to infeasible elements in a coverage domain.

The outcome of the above condition does not depend on C_2 when C_1 is `false`. When using short-circuit evaluation, condition C_2 is not evaluated if C_1 evaluates to `false`. Thus the combination $C_1 =$ `false` and $C_2 =$ `true`, or the combination $C_1 =$ `false` and $C_2 =$ `false`, may be infeasible if the programming language allows, or requires as in C, *short-circuit* evaluation.

Dependence of one decision on another might also lead to an infeasible combination. Consider, for example, the following sequence of statements.

Dependence of a decision on other might lead to infeasible elements in a coverage domain.

```
1   int  A,B,C
2   input (A,B,C);
3   if (A>10 and B>30) {
4     S1 = f1(A,B,C)
5     if (A<5 and B>10){
6       S2 = f2(A,B,C);
7   }
```

Clearly, the decision on line 5 is infeasible because the value of A cannot simultaneously be more than 10 and less than 5. However, suppose that line 3 is replaced by the following statement that has a side effect due to a call to the function `foo()`.

```
3 if(A>10 and foo ()) {
```

Given that the execution of `foo()` may lead to a modification of `A`, the condition on line 5 is feasible.

Note that feasibility is different from reachability. A decision might be reachable but not feasible, and vice versa. In the sequence above, both the decisions are reachable but the second decision is not feasible. Consider the following sequence.

A decision might be feasible but not reachable. Alternately a decision might be infeasible though reachable. Other combinations of reachability and feasibility can also be considered.

```
1   int  A,B,C
2   input (A,B,C);
3   if (A>A+1) { ← Assume that overflow causes
                   an exception.
4   S1 = f1(A,B,C)
5   if (A>5 and B>10){
6   S2 = f2(A,B,C);
7   }
```

In this case, the second decision is not reachable due to an error at line 3. It may, however, be feasible.

7.2.14 Basis path coverage

Recall that a basis set is a collection of linearly independent basis paths. The construction of basis paths is introduced in Chapter 4. Basis path coverage is simply the ratio of the number of basis paths tested to the number of feasible basis paths.

Covering all basis paths implies the coverage of all feasible decisions and hence all statements. In the presence of loops, the number of paths in a program might be simply too large to obtain a complete path coverage. Basis paths offer a practical means to select a subset of all paths to cover (se Exercise 7.29).

7.2.15 Tracing test cases to requirements

It is important that a new test case designed to cover a portion of the code be traced back to the corresponding requirements. Doing so will obviously reveal what requirement is tested by a given test case and should be the expected output.

When enhancing a test set to satisfy a given coverage criterion, it is desirable to ask the following question: *What portions of the requirements are tested when the program under test is executed against the newly added test case*? The task of relating the new test case to the requirements is known as *test trace-back*.

Trace-back has at least the following advantages. One, it assists us in determining whether or not the new test case is redundant. Second, it has the likelihood of revealing errors and ambiguities in the requirements. Third, it assists with the process of documenting tests against requirements. Such documentation is useful when requirements are modified. Modification of one or more requirements can simultaneously lead to a modification of the associated test cases thereby avoiding the need to do a detailed analysis of the modified requirements.

The next example illustrates how to associate a test, generated to satisfy a coverage criterion, to the requirements. The example also illustrates how a test generated to satisfy a coverage criterion might reveal an error in the requirements or in the program.

Example 7.27 The original test set T_1 in Example 7.23 contains only four tests, t_1, t_2, t_3, and t_4. T_2 is obtained by enhancing T_1 through the addition of t_5. We ask, "Which requirement does t_5 correspond to?" Recall that we added t_5 to cover the simple condition $x < y$ in C_2 and C_3. Going back to the requirements for P7.14, we can associate t_5 with $R_{1.1}$ and $R_{1.2}$.

One might argue that there is nothing explicit in the requirements for P7.14 that suggest a need for t_5. If one were to assume that the given requirements are correct, then indeed this argument is valid

and there is no need for t_5. However, in most software development projects, requirements are often incorrect, incomplete, and ambiguous. It is therefore desirable to extract as much information from the requirements as one possibly can, to generate new tests. Let us re examine $R_{1.2}$ in Example 7.23 reproduced below for convenience.

$R_{1.2}$: Invoke `fire-2` when ($x<y$) `and` ($z * z \leq y$) `or` (current="South").

Note that $R_{1.2}$ is not explicit about what action to take when (current="South") is `false`. Of course, as stated, the requirement does imply that `fire-2` is *not* to be invoked when (current="South") is false. Shall we go with this implication and not generate t_5? Or, shall we assume that $R_{1.2}$ is ambiguous and hence we do need a test case to test explicitly that indeed the program functions correctly when (current="South") is `false`? It is often safer to make the latter assumption. Of course, selecting the former assumption is less expensive in terms of the number of tests, the time to execute all tests, and the effort required to generate, analyze, and maintain the additional tests.

Moving further, one might ask, "What error in the program might be revealed by t_5 that will not be revealed by executing the program against all tests in T_1?" To answer this question, suppose that line 14 in P7.14 was incorrectly coded as given below while all other statements were coded as shown.

```
14  else if (x < y and (z * z ≤ y)
```

Clearly, now P7.14 is incorrect because the invocation of `fire-2` is independent of (current="South"). Execution of the incorrect program against tests in T_1 will not reveal the error because all tests lead to correct response independent of how (current="South") evaluates. However, execution of the incorrect P7.14 against t_5 causes the program to invoke `fire-3` instead of invoking `fire-2`, and hence the error is revealed. This line of argument stresses the need for t_5 generated to explicitly test (current="South").

An adamant tester might continue with the argument that t_5 was generated to satisfy the condition coverage criterion and was not based upon the requirements; it was through an examination of the code. If the code for P7.14 was incorrect due to the incorrect coding of C_3, then this test might not be generated. To counter this

argument, suppose that $R_{1.2}$ as stated earlier is incorrect; its correct version is

$R_{1.2}$: Invoke `fire-2` when ($x<y$) and ($z * z \leq y$).

Notice that now the program is correct but the requirements are not. In this case, for all tests in T_1, P7.14 produces the correct response and hence the error in requirements is not revealed. However, as mentioned in Example 7.23, t_5 is generated with the intention of covering both $x < y$ and (current==“South”). When program P7.14 is executed against t_5 it invokes `fire-3` but the requirements imply the invocation of `fire-2`. This mismatch in the program response and the requirements will likely lead to an analysis of the test results and discovery of the error in the requirements.

The above discussion justifies the association of tests t_5, t_6, t_7, t_8, and t_9 to requirement R_1. It also stresses the need for the trace-back of all tests to requirements.

7.3 Concepts From Data Flow

Control flow based adequacy criteria focus on selecting coverage domains based on the flow of control in a program. Data flow based coverage criteria focus on the flow of data for selecting coverage domains. While both strategies for determining test adequacy aid in the selection of a subset of all program paths to be tested, adequacy criteria based on data flows are generally more powerful than those based on control flows.

So far we have considered flow of control as the basis for deriving test adequacy criteria. We examined various program constructs that establish flow of control and that are susceptible to mistakes by programmers. Ensuring that all such constructs have been tested thoroughly is a major objective of test adequacy based on flow of control. However, testing all conditions and blocks of statements is often inadequate as the test does not reveal all errors.

Another kind of adequacy criteria are based on the flow of *data* through the program. Such criteria focus on definitions of data and their subsequent use and are useful in improving the tests adequate with respect to criteria based on the flow of control. The next example exposes an essential idea underlying data flow based adequacy assessment.

Example 7.28 The following program inputs integers x and y, and outputs z. The program contains two simple conditions. As shown, there is an error in this program at line 8. An MC/DC adequate test follows the program.

Program P7.15

```
1   begin
2     int x,y; floatz;
3     input (x,y);
```

```
4     z=0;
5     if (x!=0)
6       z=z+y;
7     else  z=z-y;
8     if (y!=0) ← This condition should be (y!=0 and x!=0)
9       z=z/x;
10     else z=z*x;
11     output(z);
12   end
```

Test	x	y	z
t_1	0	0	0.0
t_2	1	1	1.0

The two test cases above cover both conditions in P7.15. Note that the program initializes z on line 4 and then adjusts its value in subsequent statements depending on the values of x and y. Test t_1 follows a path that traverses updates of z on line 7 followed by another on line 10. Test t_2 follows a different path that traverses updates of z on line 6 followed by another update on line 9. These two tests cause the value of z, set at at line 4, to be used at lines 6 and 7 in the expression on the corresponding right hand sides.

Just as z is defined at line 4, it is also defined subsequently at lines 6, 7, 9, and 10. However, the two MC/DC adequate tests do not force the definition of z at line 6 to be used at line 9. A divide-by-zero exception would occur if the value of z were allowed to flow from line 6 to line 9. Though this exception is not caused by the value of z, forcing this path causes the program to fail.

An MC/DC adequate test does not force the execution of this path and hence the error is not revealed. However, the error in this program would be revealed if the test is required to ensure that each *feasible* definition and use pair for z is executed. One such test follows.

An MC/DC adequate test set might not cover all feasible data flows in a program. Thus, any errors due to incorrect data flows might miss detection by a test that is MC/DC adequate but not adequate with respect to a data flow based adequacy criterion.

Test	x	y	z	*def-use pairs covered
t_1	0	0	0.0	(4,7), (7,10)
t_2	1	1	1.0	(4,6), (6,9)
t_3	0	1	0.0	(4,7), (7,9)
t_4	1	0	1.0	(4,6), (6,10)

*In the pair (l_1, l_2), z is defined at line l_1 and used in line l_2.

It is easy to see that an LCSAJ adequate test set would have also revealed the error in P7.15, though an MC/DC adequate test might not. In the

remainder of this section, we will offer examples where an LCSJ adequate test will also not guarantee that an error is revealed but a dervied test set based on data flow will.

A variable is considered defined when the result of a computation is assigned to it. In OO languages, a name is also considered defined when an object is assigned to it. For example, using the "new" command in Java a new object can be created and assigned to a name. At this point in the code the name is considered defined.

A variable could be defined through the use of statements such as assignment state, an input statement, via parameter transfer. A variable is considered to be used when it appears in the right hand side of an assignment, or in a condition or in an output statement.

7.3.1 Definitions and uses

A program written in a procedural language, such as C and Java, contains variables. Variables are defined by assigning values to them and are used in expressions. An assignment statement such as

```
x = y + z;
```

defines the variable x. This statement also makes use of variables y and z. The following assignment statement

```
x = x + y;
```

defines x as well as uses it in the right hand side expression. Declarations are assumed to be definitions of variables. For example, the declaration

```
int x, y, A[10];
```

defines three variables x, y, A[10]. An input function, such as a `scanf` in C, also serves to define one or more variables. For example, the statement

```
scanf("%d %d", &x, &y);
```

defines variables x and y. Similarly, the C statement

```
printf("Output: %d \ n", x + y);
```

uses, or *refers to*, variables x and y. A parameter x, passed as call-by-value to a function, is considered as a use of, or a reference to, x. A parameter x passed as call-by-reference serves as a definition and use of x. Consider the following sequence of statements that use pointers.

```
z = &x;
y = z + 1;
*z = 25;
y = *z + 1;
```

The first of the above statements defines a pointer variable z, the second defines y and uses z, the third defines x through the pointer variable z, and the last defines y and uses x accessed through the pointer variable z. Arrays are also tricky. Consider the following declaration and two statements in C:

```
int A[10];
A[i]=x+y;
```

The first statement defines variable A. The second statement defines A and uses i, x, and y. One might also consider the second statement as defining only A[i] and not the entire array A. The choice of whether to consider the entire array A as defined or the specific element depends upon how stringent is the requirement for coverage analysis (see Exercise 7.28).

7.3.2 C-use and p-use

Uses of a variable that occur within an expression as part of an assignment statement, in an output statement, as a parameter within a function call, and in subscript expressions are classified as *c-use*, where the "c" in c-use stands for *computational*. There are five c-uses of x in the following statements.

A use of a variable is considered to be c-use (or computational use) when it appears inside a computation such as in an arithmetic expression. When the use appears in a condition the use is considered a p-use (or predicate use).

```
z=x+1;
A[x-1]= B[2];
foo(x*x);
output(x);
```

An occurrence of a variable in an expression used as a condition in a branch statement, such as an `if` and a `while`, is considered a p-use. The "p" in p-use stands for *predicate*. There are two p-uses of z in the statements below.

```
if(z>0){output (x)};
while(z>x){...};
```

It might be confusing to classify some uses. For example, in the statement:

```
if(A[x+1]>0){output (x)};
```

the use of A is clearly a p-use. However, is the use of x within the subscript expression also a p-use? Or, is it a c-use? The source of confusion is the position of x within the `if` statement. x is used in an expression that computes the index value for A; it does not directly occur inside the `if` condition such as in `if(x>0){...}`. Hence, one can argue both ways: this occurrence of x is a c-use because it does not affect the condition directly, or that this occurrence of x is a p-use because it occurs in the context of a decision.

7.3.3 Global and local definitions and uses

A variable might be defined in a block, used, and redefined within the same block. Consider a block containing three statements:

```
p=y+z;
x=p+1;
p=z*z;
```

When a variable is defined multiple times inside a basic block, only the effect of the last definition "flows out" of the defining block. This last definition is also known as global while the ones prior to it are considered local.

This block defines `p`, uses it, and redefines it. The first definition of `p` is *local*. This definition is *killed*, or *masked*, by the second definition within the same block and hence its value does not survive beyond this block. The second definition of `p` is a *global* definition as its value survives the block within which it is defined and is available for use in a subsequently executed block. Similarly, the first use of `p` is a *local* use of a definition of `p` that precedes it within the same block.

The c-uses of variables `y` and `z` are *global* uses because the definitions of the corresponding variables do not appear in the same block preceding their use. Note that the definition of `x` is also global. In the remainder of this chapter, we will be concerned only with the global definitions and uses. Local definitions and uses are of no use in defining test adequacy based on data flows.

The terms "global" and "local" are used here in the context of a basic block within a program. This is in contrast to the traditional uses of these terms where a global-variable is one declared outside, and a local variable within a function or a module.

7.3.4 Data flow graph

A data flow graph of a program, also known as def-use graph, captures the flow of definitions across basic blocks in a program. It is similar to a control flow graph of a program in that the nodes, edges, and all paths through the control flow graph are preserved in the data flow graph.

A data flow graph of a program captures the data flows, i.e. the relationships between all the definitions and uses in a program. The graph can be derived from the control flow graph of a program.

A data flow graph of a program can be derived from its control flow graph. To understand how, let $G = (N, E)$ be the control flow graph of program P, where N is the set of nodes and E the set of edges in the graph. Recall that each node in the control flow graph corresponds to a distinct basic block in P. We denote these blocks by $b_1, b_2, \ldots, b_k$, assuming that P contains $k > 0$ basic blocks.

Let def_i denote the set of variables defined in block i. A variable declaration, an assignment statement, an input statement, a call-by-reference parameter are program constructs that define variables. Let c-use_i denote the set of variables that have a c-use in block i, and p-use_i the set of variables that occur as a p-use in block i. The definition of variable x at node i is denoted by $d_i(x)$. Similarly, the use of variable x at node i is denoted by $u_i(x)$. Recalling that we consider only the global definitions and uses, consider the following basic block b that contains two assignments and a function call.

```
p=y+z;
foo(p+q, number); // Parameters passed by value.
A[i]=x+1;
if(x>y){...};
```

For this basic block, we obtain $\text{def}_b = \{\texttt{p}, \texttt{A}\}$, $\text{c-use}_b = \{y, z, p, q, x, \texttt{number}, i\}$, and $\text{p-use}_b = \{x, y\}$. We follow the procedure below to construct a data flow graph given program P and its control flow graph.

Step 1 Construct def_i, c-use_i, and p-use_i for each basic block i in P.
Step 2 Associate each node i in N with def_i, c-use_i, and p-use_i.
Step 3 For each node i that has a non-empty p-use set and ends in condition C, associate edges (i, j) and (i, k) with C and $!C$, respectively, given that edge (i, j) is taken when the condition is true and (i, k) taken when the condition is false.

The next example applies the procedure given above to construct a data flow graph in Figure 7.4.

Example 7.29 First we compute the definitions and uses for each block in Figure 7.4 and associate them with the corresponding nodes and edges of the control flow graph. The *def* and *use* sets for the five blocks in this program are given below. Note that the infeasible block 4 does not affect the computation of these sets.

The def-set of a block is the set of all variables defined in that block. The c-use and the p-use sets of a block include, respectively, all the c- and p-uses in that block.

Node (or Block)	def	c-use	p-use
1	$\{x, y, z\}$	$\{\ \}$	$\{x, y\}$
2	$\{z\}$	$\{x\}$	$\{y\}$
3	$\{z\}$	$\{z\}$	$\{\ \}$
4	$\{z\}$	$\{x\}$	$\{\ \}$
5	$\{\ \}$	$\{z\}$	$\{\ \}$

Using the def and use sets derived above, and the flow graph in Figure 7.4, we draw the data flow graph of P7.4 in Figure 7.6. Comparing the data flow graph with the control flow graph in Figure 7.4, we note that nodes are represented as circles with block numbers on the inside, each node is labeled with the corresponding def and use sets, and each edge is labeled with a condition. We have omitted the p-use set when it is empty, such as for nodes 3, 4, and 5. p-uses are often associated with edges in a data flow graph.

As shown in Figure 7.6 we associate p-uses with nodes that end in a condition, such as in an `if` or a `while` statement. Each variable in this p-use set also occurs along the two edges out of its corresponding node. Thus variables x and y belong to the p-use set for node 1 and also appear in the conditions attached to the outgoing edges of node 1.

As shown in Figure 7.6(b), the `Start` node may be omitted from the data flow graph if there is exactly one program block with no edge entering.

Figure 7.6 Data flow graph of 7.8 (a) `Start` and `Exit` nodes shown explicitly. No def and use sets are associated with these nodes. (b) Nodes 1 and 5 serve as start and end nodes, respectively. Node 3 is shaded to emphasize that it is unreachable.

Similarly, the `End` node may be omitted in Figure 7.6 and use nodes 1 and 5 as the start and end nodes, respectively.

A def-clear path from a definition of a variable, say x, to its use is a path along which x is not redefined.

Of course, multiple definitions of a variable could be live at a specific point in the program. However, each of these definitions will flow to this point along a different path. Hence, at most one definition of a variable can be live when control arrives at a specific point in the program where that variable is used.

7.3.5 Def-clear paths

A data flow graph can have many paths. One class of paths of particular interest is known as the *def-clear* path. Suppose that variable x is defined at node i and there is a use of x at node j. Consider path $p = (i, n_1, n_2, \ldots, n_k, j), k \geq 0$, that starts at node i, ends at node j, and nodes i and j do not occur along the subpath $n_1, n_2, \ldots, n_k$. p is a def-clear path for variable x defined at node i and used at node j if x is not defined along the subpath $n_1, n_2, \ldots, n_k$. In this case, we also say that the definition of x at node i, i.e. $d_i(x)$, is *live* at node j.

Note that several definitions of variable x may be live at some node j where x is used, but along different paths. Also, at most one definition could be live when control does arrive at node j where x is used.

Example 7.30 Consider the data flow graph for Program P7.16 shown in Figure 7.7. Path $p = (1, 2, 5, 6)$ is def-clear with respect to $d_1(x)$ and $u_6(x)$. Thus $d_1(x)$ is live at node 6. Path p is not def-clear with respect to $d_1(z)$ and $u_6(z)$ due to the existence of $d_5(z)$. Also, path p is def-clear with respect to d_1 `(count)` and u_6 `(count)`.

Path $q = (6, 2, 5, 6)$ is def-clear with respect to d_6 `(count)` and d_6 `(count)` (see Exercise 7.32). Variables y and z are used at

node 4. It is easy to verify that definitions $d_1(y)$, $d_6(y)$, $d_1(z)$, $d_5(z)$ are live at node 4.

Program P7.16

```
1   begin
2    float x,y,z=0.0;
3    int count;
4    input (x,y,count);
5    do  {
6     if (x≤0) {
7      if (y≥0) {
8        z=y*z+1;
9      }
10    }
11    else{
12     z=1/x;
13    }
14    y=x*y+z;
15    count=count-1;
16   }//  End of loop body
17   while (count>0)
18   output (z);
19  end
```

Node	Lines
1	1, 2, 3, 4
2	5, 6
3	7
4	8, 9, 10
5	11, 12, 13
6	14, 15, 16
7	17, 18

Figure 7.7 Data flow graph of Program P7.16.

7.3.6 Def-use pairs

We saw earlier that a def-use pair captures a specific definition and its use for a variable. For example, in P7.4, the definition of x at line 4 and its

The definition of a variable at some point l1 in a program and its use at some point l2 in the same program constitute a def-sue pair. Note that l1 and l2 could be the same statements in a program.

use at line 9 together constitute a def-use pair. The data flow graph of a program contains all def-use pairs for that program.

We are interested in two types of def-use pairs: those that correspond to a definition and its c-use, and those that correspond to a definition and its p-use. Such def-use pairs are captured in sets named **dcu** and **dpu**.

A def-use pair is placed in the dcu set if the use in the pair is a c-use, else it is placed in set dpu. Set dcu consists of all locations in a program where a specific definition of a variable has a c-use. For each definition of a variable, set dpu contains edges (k, l) such that there is a path from where the variable is defined to location l; location k is where the p-use occurs.

There is one **dcu** and one **dpu** set for each variable definition. For $d_i(x)$, **dcu**($d_i(x)$) is the set of all nodes j such that there exists $u_j(x)$ and there is a def-clear path with respect to x from node i to node j. **dcu**(x, i) is an alternate notation for **dcu**($d_i(x)$).

When $u_k(x)$ occurs in a predicate, **dpu**($d_i(x)$) is the set of edges (k, l) such that there is a def-clear path with respect to x from node i to node l. Note that the number of elements in a **dpu** set will be a multiple of two. **dpu**(x, i) is an alternate notation for **dpu**($d_i(x)$).

Example 7.31 We compute the **dcu** and **dpu** sets for the data flow graph in Figure 7.7. To begin, let us compute **dcu**(x, 1) which corresponds to the set of all c-uses associated with the definition of x at node 1. We note that there is a c-use of x at node 5 and a def-clear path (1, 2, 5) from node 1 to node 5. Hence node 5 is included in **dcu**(x, 1). Similarly, node 6 is also included in this set. Thus we get **dcu**(x, 1)={5, 6}.

Next, determine **dpu**(x, 1). There is a p-use of x at node 2 with outgoing edges (2, 3) and (2, 5). The existence of def-clear paths from node 1 to each of these two edges is easily seen in the figure. There is no other p-use of x and hence we get **dpu**(x, 1)={(2, 3), (2, 5)}. The remaining **dcu** and **dpu** sets are given below.

Variable (v)	**Defined at node** (n)	**dcu** (v, n)	**dpu** (v, n)
x	1	{5, 6}	{(2, 3), (2, 5)}
y	1	{4, 6}	{(3, 4), (3, 6)}
y	6	{4, 6}	{(3, 4), (3, 6)}
z	1	{4, 6, 7}	{ }
z	4	{4, 6, 7 }	{ }
z	5	{4, 6, 7}	{ }
count	1	{6}	{(6, 2), (6, 7)}
count	6	{6}	{(6, 2), (6, 7)}

7.3.7 Def-use chains

A def-use pair consists of a definition of a variable in some block and its use in another block in the program. The notion of def-use pair can be extended to a sequence of alternating definitions and uses of variables. Such an alternating sequence is known as a *def-use chain*, or a *k-dr*

interaction. The nodes along this alternating sequence are distinct. The k in *k-dr* interaction denotes the length of the chain and is measured in the number of nodes along the chain which is one more than the number of def-use pairs along the chain. The letters d and r refer to, respectively, *definition* and *reference*. Recall that we use the terms "reference" and "uses" synonymously.

A def-use chain a an alternating sequence of def-use pairs for a specific variable. Another name for a def-use chain is a k-dr interaction, k being the length of the chain.

Example 7.32 In Program P7.16, we note the existence of $d_1(z)$ and $u_4(z)$. Thus the def-use interaction for variable z at nodes 1 and 4 constitutes a *k-dr* chain for $k = 2$. This chain is denoted by the pair (1, 4).

A longer chain (1, 4, 6) is obtained by appending $d_4(z)$ and $u_6(z)$ to (1, 4). Note that there is no chain for $k > 3$ that corresponds to alternating def-uses for z. Thus, for example, path (1, 4, 4, 6) for $k = 4$ corresponding to the def-use sequence ($d_1(z)$, $u_4(z)$, $d_4(z)$, $u_6(z)$) is not a valid *k-dr* chain due to the repetition of node 4.

The chains in the above example correspond to the definition and use of the same variable z. *k-dr* chains could also be constructed from alternating definitions and uses of variables not necessarily distinct. In general, a *k-dr* chain for variables $x_1, x_2, \ldots, x_{k-1}$ is a path $(n_1, n_2, \ldots, n_k)$ such that there exist $d_{n_i}(X_i)$ and $u_{n_{i+1}}(x_i)$ for $1 \leq i < k$, and there is a definite clear path from n_i to n_{i+1}. This definition of a *k-dr* chain implies that variable x_{i+1} is defined in the same node as variable x_i. A chain may also include the use of a variable in a predicate.

Example 7.33 Once again, let us refer to Program P7.16 and consider variables y and z. The triple (5, 6, 4) is a *k-dr* chain of length 3. It corresponds to the alternating sequence ($d_5(z)$, $u_6(z)$, $d_6(y)$, $u_4(y)$). Another chain (1, 4, 6) corresponds to the alternating sequence ($d_1(y)$, $u_4(y)$, $d_4(z)$, $u_6(z)$).

While constructing *k-dr* chains, it is assumed that each branch in the data flow graph corresponds to a simple predicate. In the event a decision in the program is a compound predicate, such as (`x < y`) `and` (`z < 0`), it is split into two nodes for the purpose of *k-dr* analysis (see Exercise 7.39).

7.3.8 A little optimization

The coverage of one def-use might imply the coverage of another. This fact can be used to reduce the number of def-uses to be covered.

Often one can reduce the number of def-uses to be covered by a simple analysis of the flow graph. To illustrate what we mean, refer to the def-use graph in Figure 7.7. From this graph we discovered that **dcu**(y, 1)={4, 6}. We also discovered that **dcu**(z, 1)=(4, 6, 7). We now ask: *Will the coverage of* **dcu***(z, 1) imply the coverage of* **dcu***(y, 1)={4, 6}?* Path (1, 2, 3, 4) must

be traversed to cover the c-use of z at node 4 corresponding to its definition at node 1. However, traversal of this path also implies the coverage of the c-use of y at node 4 corresponding to its definition at node 1.

Similarly, path (1, 2, 3, 6) must be traversed to cover the c-use of z at node 6 corresponding to its definition at node 1. Once again, the traversal of this path implies the coverage of the c-use of y at node 6 corresponding to its definition at node 1. This analysis implies that we need not consider **dcu**(y, 1) while generating tests to cover all c-uses; **dcu**(y, 1) will be covered automatically if we cover **dcu**(z, 1).

Arguments, analogous to the ones above, can be used to show that **dcu**(x, 1) can be removed from consideration when developing tests to cover all c-uses. This is because **dcu**(x, 1) will be covered when **dcu**(z, 5) is covered. Continuing in this manner, one can show that **dcu**(`count`, 1) can also be ignored. This leads us to a minimal set of c-uses to be covered as shown in the table below. Note that the total number of c-uses to consider has now been reduced to 12 from 17. The total number of p-uses to consider has been reduced to 4 from 10. Similar analysis for removing p-uses from consideration is left as an exercise (see Exercise 7.34).

Variable (`v`)	**Defined at node** (`n`)	**dcu** (`v`, `n`)	**dpu** (`v`, `n`)
y	6	{4, 6}	{(3, 4), (3, 6)}
z	1	{4, 6, 7}	{ }
z	4	{4, 6, 7 }	{ }
z	5	{4, 6, 7}	{ }
`count`	6	{6}	{(6, 2), (6, 7)}

In general, it is difficult to carry out the type of analysis illustrated above for programs of non-trivial size. Usually test tools, such as χSUDS, automatically do such analysis and minimize the number of c- and p-uses to consider while generating tests to cover all c- and p-uses. Note that this analysis does not remove any c- and p-use from the program, it simply removes some of them from consideration because they are automatically covered by tests that cover some other c- and p-uses.

7.3.9 Data contexts and ordered data contexts

A data environment is the set of all variables whose definitions are live at a node in a data flow graph.

Let n be a node in a data flow graph. Each variable used at n is known as an *input* variable of n. Similarly, each variable defined at n is known as an *output* variable of n. The set of all live definitions of all input variables of node n is known as the *data environment* of node n and is denoted by $DE(n)$.

Let $X(n) = \{x_1, x_2, \ldots, x_k\}$ be the set of input variables required for evaluation of any expression, arithmetic or otherwise, at node n in data flow

graph F. Let $x_j^{i_j}$ denote the i_jth definition of x_j in F. An *elementary data context* of node n, denoted as $EDC(n)$, is the set of definitions $\{x_1^{i_1}, x_2^{i_2} \ldots, x_k^{i_k}\}$, $i_k \geq 1$, of all variables in $X(n)$ such that each definition is live when control arrives at node n. The *data context* of node n is the set of all its elementary data contexts and is denoted by $DC(n)$. Given that the i_jth definition of x_j occurs at some node k, we shall refer to it as $d_k(x_j)$.

A data context is the set of all elementary data contexts for a node in a data flow graph.

Example 7.34 Let us examine node 4 in the data flow graph of Figure 7.7. The set of input variables of node 4, denoted by $X(4)$, is $\{y, z\}$. The data environment for node 4, denoted by $DE(4)$, is the set $\{d_1(y), d_6(y), d_1(z), d_4(z), d_5(z)\}$.

Recall that each definition in $DE(4)$ is live at node 4. For example, $d_4(z)$ is live at node 4 because there exists a path from the start of the program to node 4, and a path from node 4 back to node 4 without a redefinition of z until it has been used at node 4. This path is (1, 2, 3, $\underline{4}$, 6, 2, 3, $\underline{4}$). The first emphasized occurrence of node 4 is the one where z is defined and the second emphasized occurrence of node 4 is where this definition is live and used.

The data context $DC(4)$ for node 4, and all other nodes in the graph, is given in the following table. Notice that the data context is defined only for nodes that contain at least one variable c-use. Thus, while determining data contexts, we exclude nodes that contain only a predicate. Nodes 1, 2, and 3 are three such nodes in Figure 7.7. Also, we have omitted some infeasible data contexts for node 6 (see Exercise 7.44).

Node k	Data context DC(k)
1	None
2	None
3	None
4	$\{(d_1(y), d_1(z)), (d_6(y), d_4(z)), (d_6(y), d_5(z))\}$
5	$\{(d_1(z))\}$
6	$\{ (d_1(z), d_1(y), d_1(z), d_1(\texttt{count})), (d_1(z), d_1(y), d_5(z), d_1(\texttt{count}))),$ $(d_1(z), d_1(y), d_4(z), d_1(\texttt{count})), (d_1(z), d_6(y), d_5(z), d_6(\texttt{count})),$ $(d_1(z), d_6(y), d_4(z), d_6(\texttt{count}))\}$
7	$\{(d_1(z)), (d_4(z)), (d_5(z))\}$

The definitions of variables in an elementary data context occur along a program path in an order. For example, consider the elementary data context $\{d_6(y), d_4(z)\}$ for node 4 in Figure 7.7. In the second occurrence of node 4 along the path (1, 2, 3, 4, 6, 2, 3, $\underline{4}$, 6, 7), $d_4(z)$ occurs first followed by $d_6(y)$. The sequence $d_6(z)$ followed by $d_6(y)$ is not feasible when control

arrives at node 4. Hence only one sequence, i.e. $d_6(y)$ followed by $d_4(z)$, is possible for node 4.

Now consider the second occurrence of node 6 along the path (1, 2, 3, 6, 2, 5, 6, 7). The sequence of definitions of input variables of node 6 is $d_1(x)$ followed by $d_6(y)$, $d_6(\texttt{count})$, and lastly $d_5(z)$. Next, consider again the second occurrence of node 6 but now along the path (1, 2, 5, 6, 2, 3, 6). In this case, the sequence of definitions is $d_1(x)$ followed by $d_5(z)$, $d_6(y)$, and lastly $d_6(\texttt{count})$. These two sequences are different in that y and z are defined in a different sequence.

An ordered elementary data contexts imposes an ordering on the elements of an elementary data context. An ordered data context is the set of all elementary ordered data contexts.

The above examples lead to the notion of an *ordered elementary data context* of node n abbreviated as *OEDC(n)*. An ordered elementary data context for node n in a data flow graph consists of a set of ordered sequence of definitions of the input variables of n. An *ordered data context* for node n is the set of all ordered elementary data contexts of node n denoted by $ODC(n)$.

Example 7.35 Consider Program P7.17. The data flow graph for this program appears in Figure 7.8. The set of input variables for node 6 is $\{x, y\}$. The data context $DC(6)$ for node 6 consists of four elementary data contexts given below.

DC(6)=$\{d_1(x), d_1(y)), (d_3(x), d_1(y)), (d_1(x), d_4(y)), (d_3(x), d_4(y))\}$

The ordered data context for the same node is given below.

$ODC(6)= \{(d_1(x), d_1(y)), (d_1(y), d_3(x)), (d_1(x), d_4(y)), (d_3(x), d_4(y)), (d_4(y), d_3(x))\}$

Notice that due to the inclusion of ordered elementary data contexts, $ODC(6)$ has all items from $DC(6)$ and one additional item that is not in $DC(6)$.

Program P7.17

```
1   begin
2    int x,y,z,count;
3    input (x,y,count);
4    while (count>0) {
5     if (x<y)
6      x= foo1(x-y);
7     else
8      y= foo1(x+y);
9     count=count-1;
10   }
11   z= foo2(x, y);
12   output (z);
13  end
```

Node	Lines
1	1, 2, 3, 4
2	5
3	6
4	7, 8
5	9, 10
6	11, 12, 13

Figure 7.8 Data flow graph of Program P7.17 to illustrate ordered data context.

7.4 Adequacy Criteria Based on Data Flow

To be able to construct test adequacy criteria based on sets **dpu** and **dcu**, we first define some notions related to coverage. In the definitions below we assume that the data flow graph of P contains k nodes where nodes n_1 and n_k represent the start and end nodes, respectively. Some node s in the data flow graph of program P is considered covered by test case t when the following complete path is traversed upon the execution of P against t,

Test adequacy criteria based on data flows can be defined using a data flow graph.

$$(n_{i_1}, n_{i_2}, \ldots, n_{i_{m-1}}, n_{i_m})$$

and $s = n_{i_j}$ for some $1 \leq j \leq m$ and $m \leq k$, i.e. node s lies on the path traversed. Similarly, an edge (r, s) in the data flow graph of P is considered covered by t upon the execution of P against t, when the path listed above is traversed and $r = n_{i_j}$ and $s = n_{i_{j+1}}$ for $1 \leq j \leq (m-1)$.

Let CU and PU denote, respectively, the total number of c-uses and p-uses for all variable definitions in P. Let $v = \{v_1, v_2, \ldots, v_n\}$ be the set of variables in P. d_i is the number of definitions of variable v_i for $1 \leq i \leq n$. CU and PU are computed as follows.

$$\text{CU} = \Sigma_{i=1}^{n} \Sigma_{j=1}^{d_i} |\mathbf{dcu}(v_i, n_j)|$$
$$\text{PU} = \Sigma_{i=1}^{n} \Sigma_{j=1}^{d_i} |\mathbf{dpu}(v_i, n_j)|$$

Recall that $|S|$ denotes the number of elements in set S. For the program in Example 7.31, we get CU=17 and PU=10.

7.4.1 c-use coverage

A c-use of a variable x defined at location l1 and used at location l2 at is considered covered when control flows along a path from l1 to l2 without any redefinition of x.

Let z be a node in **dcu**(x, q), i.e. node z contains a c-use of variable x defined at node q (see Figure 7.9(a)). Suppose that program P is executed against test case t and the (complete) path traversed is:

$$p = (n_1, n_{i_1}, \ldots, n_{i_l}, n_{i_{l+1}}, \ldots, n_{i_m}, n_{i_{m+1}} \ldots n_k), \text{ where } 2 \leq i_j < k \text{ for } 1 \leq j \leq k$$

This c-use of variable x is considered covered if $q = n_{i_l}$ and $s = n_{i_m}$ and $(n_{i_l}, n_{i_{l+1}}, \ldots, n_{i_m})$ is a def-clear path from node q to node z. All c-uses of variable x are considered covered if each node in **dcu**(x, q) is covered during one or more executions of P. All c-uses in the program are considered covered if all c-uses of all variables in P are covered. We now define test adequacy criterion based on c-use coverage.

Figure 7.9 Paths for c-use and p-use coverage. Dotted arrows connecting two nodes indicate a path along which there might be additional nodes not shown. (a) Path (`Start`, ..., q, k, ..., z, ..., `End`) covers the c-use at node z of x defined at node q given that (k, ..., z) is def-clear with respect to x. (b) The p-use of x at node z is covered when paths (`Start`, ..., q, k, ..., z, r, ..., `End`) and (`Start`, ..., q, k, ..., z, s, ..., `End`), shown by dotted lines, have been traversed, and (k, ..., z) are def-clear with respect to x.

C-use coverage:
The c-use coverage of T with respect to (P, R) is computed as

$$\frac{CU_c}{CU - CU_f}$$

where CU_C is the number of c-uses covered and CU_f the number of infeasible c-uses. T is considered adequate with respect to the c-use coverage criterion if its c-use coverage is 1.

A test set is considered adequate with respect to the c-use criterion when it covers all feasible c-uses in the program under test.

Example 7.36 Suppose that Program P7.16 is under test. The data flow graph of this program is shown in Figure 7.7. We want to devise a test t_c which covers node 6 in **dcu**(z, 5), i.e. the desired test must cover the c-use of the definition of x at node 1. Consider the following test case.

$t_c : < x = 5, y = -1, \texttt{count} = 1 >$

The complete path traversed when Program P7.16 is executed against t_c is: (1, 2, 5, 6, 7). Clearly, this path contains node 5 at which z is defined, node 6 where it has a c-use, and subpath (5, 6) is a def-clear path with respect to this definition of z. Hence t_c covers node 6, an element of **dcu**(z, 6).

Note that t_c also covers node 7 but not node 4, which also contains a c-use of this definition of z. Considering that there are 12 c-uses in all, and t_c covers only 2 of them, the c-use coverage for $T = \{t_c\}$ is $2/12 = 0.167$. Clearly, T is not adequate with respect to the c-use criterion for Program P7.16.

7.4.2 p-use coverage

Let (z, r) and (z, s) be two edges in **dpu**(x, q), i.e. node z contains a p-use of variable x defined at node q (see Figure 7.9(b)). Suppose that the following complete path is traversed when P is executed against some test case t_p.

A p-use of a variable x defined at location l1 and used at location l2 in a predicate at location l2 is considered covered when control flows along a path from l1 to l2 and then to each of the destinations of the predicate. Of course, each p-use coverage requires the traversal of at least two paths.

$(n_1, n_{i_2}, \ldots, n_{i_l}, n_{i_{l+1}}, \ldots, n_{i_m}, n_{i_{m+1}}, \ldots, n_k)$ where $2 \le i_j < k$ for $1 \le j \le k$

The following condition must hold for edge (z, r) for the p-use at node z of x defined at node q, to be covered.

$q = n_{i_l}$, $z = n_{i_m}$, $r = n_{i_{m+1}}$ and $(n_{i_l}, n_{i_{l+1}}, \ldots, n_{i_m}, n_{i_{m+1}})$ is a def-clear path with respect to x.

Similarly, the following condition must hold for edge (z, s).

$q = n_{i_l}$, $z = n_{i_m}$, $s = n_{i_{m+1}}$, and $(n_{i_l}, n_{i_{l+1}}, \ldots, n_{i_m}, n_{i_{m+1}})$ is a def-clear path with respect to x.

The p-use of x at node z is considered covered when the two conditions mentioned above are satisfied in the same or different executions of P. We can now define test adequacy criterion based on p-use coverage.

P-use coverage:
The p-use coverage of T with respect to (P, R) is computed as

$$\frac{PU_c}{PU - PU_f}$$

where PU_C is the number of p-uses covered and PU_f the number of infeasible p-uses. T is considered adequate with respect to the p-use coverage criterion if its p-use coverage is 1.

A test set is considered adequate with respect to the p-use criterion when it covers all feasible p-uses in the program under test.

Example 7.37 Once again, suppose that Program P7.16 is under test. Consider the definition of y at node 6. Our objective now is to cover the p-use of this definition at node 3. This requires a test that must cause control to reach node 6 where y is defined. Without redefining y, the control must now arrive at node 3, where y is used in the `if` condition.

The edge labeled $\mathtt{y} \geq 0$ is covered if taken, though the p-use is yet to be covered. Subsequently, control should take the edge labeled $\mathtt{y} < 0$ during the same or in some other execution. Consider the following test case designed to achieve the coverage objective.

$t_p : < x = -2, y = -1, \mathtt{count} = 3 >$

A careful examination of Program P7.16 and its data flow graph reveals that the following path is traversed upon the execution of the program against t_p.

$p = < 1, 2, 3, 6_1, 2, 3_1, 4_1, 6_2, 2, 3_2, 6_3, 7 >$

The first two instances of subscripted node 6 correspond to the definitions of y in the first and the second iterations, respectively, of the loop. The remaining subscripted nodes indicate edges along which the values of y are propagated. An explanation follows.

Variable y is defined at node 6 when subpath $< 1, 2, 3, 6_1 >$ is traversed. This value of y is used in the decision at node 3 when $< 6_1, 2, 3_1, 4_1 >$ is traversed. Note that edge (3, 4) is now covered as there is a def-clear path $< 2, 3 >$ from node 6 to edge (3, 4).

Next, path $< 6_2, 2, 3_2, 6_3, 7 >$ is traversed where y is defined once again at node 6. This definition of y propagates to edge (3, 6) via the def-clear path $< 2, 3_2, 6_2 >$. The control now moves to node 7 and the program execution

terminates. Test t_p has successfully covered **dpu**(y, 6). In fact all the p-uses in Program P7.16 are covered by the test set $T = \{t_c, t_p\}$. Thus the p-use coverage of T is 1.0 and hence this test set is adequate with respect to the p-use coverage criterion (see Exercise 7.35).

7.4.3 All-uses coverage

The all-uses coverage criterion is obtained by combining the c-use and p-use criteria. The all-uses criterion is satisfied when all c-uses and all p-uses have been covered. The definition can be stated as follows in terms of a coverage formula.

All-uses coverage:
The all-uses coverage of T with respect to (P, R) is computed as

$$\frac{(CU_c + PU_c)}{(CU + PU) - (CU_f + PU_f)}$$

where CU is the total c-uses, CU_c is the number of c-uses covered, PU_c is the number of p-uses covered, CU_f the number of infeasible c-uses, and PU_f the number of infeasible p-uses. T is considered adequate with respect to the all-uses coverage criterion if its c-use coverage is 1.

A test set is considered adequate with respect to the all-uses criterion when it covers all feasible c- and p-uses in the program under test.

Example 7.38 For Program P7.16 and the test cases taken from Examples 7.36 and 7.37, the test set $T = \{t_c, t_p\}$ is adequate with respect to the all-uses criterion.

7.4.4 k-dr chain coverage

Consider a *k-dr* chain $C = (n_1, n_2, \ldots, n_k)$ in program P such that $d_{n_i}(x_i)$ and $u_{n_{i+1}}(x_i)$ for $1 \leq i < k$. We consider two cases depending on whether or not node n_k ends in a predicate. Given that node n_k does not end in a predicate, chain C is considered covered by test set T only if upon the execution of P against T, the following path is traversed:

$$(\texttt{Start}, \ldots, n_1, p_1, n_2, \ldots, p_{k-1} n_k, \ldots, \texttt{End})$$

where each p_i for $1 \leq i < k$ is a definition clear subpath from node n_i to node n_{i+1} with respect to variable x_i. If node n_k ends in a predicate, then C is considered covered only if upon the execution of P against T, the following paths are traversed:

$$(\texttt{Start}, \ldots, n_1, p_1, n_2, \ldots, p_{k-1} n_k, r, \ldots, \texttt{End})$$
$$(\texttt{Start}, \ldots, n_1, p'_1, n_2, \ldots, p'_{k-1} n_k, s, \ldots, \texttt{End})$$

where p_i and p'_is denote definition clear subpaths for variable x_i and nodes r and s are immediate successors of node n_k. The above condition ensures that the decision containing the last use of a variable along the chain is covered, i.e. both branches out of the decision are taken. As shown in Example 7.40, it is possible to traverse both paths in one execution.

While determining *k-dr* chains, it is assumed that each branch in a data flow graph corresponds to a simple predicate. In the event a decision in the program is a compound predicate, such as (`x < y`) `and` `z < 0`), it is split into two simple predicates. Splitting ensures that each decision node in the corresponding data flow graph will contain an atomic predicate (see Exercise 7.39).

When the first definition or the last reference of a variable is inside a loop, at least two iteration counts must be considered for the loop. We consider these two iteration counts to be 1 and 2, i.e. the loop is iterated once and exited, and in the same or a different execution, iterated twice and exited.

Example 7.39 Consider the 3-dr chain $C = (1, 5, 7)$ from Example 7.32 for Program P7.16. Chain C corresponds to the alternating defuse sequence ($d_1(\mathtt{x})$, $u_5(\mathtt{x})$, $d_5(z)$, $d_7(\mathtt{z})$). The following test covers C.

$$T = \{t : < \mathtt{x} = 3, \mathtt{y} = 2, \mathtt{count} = 1 >\}$$

Example 7.40 Consider the 3-dr chain $C = (1, 4, 6)$ from Example 7.32 for Program P7.16. We note that C ends at node 6 that ends in a predicate. Nodes 2 and 7 are immediate successors of node 6. Hence, to cover C, the following subpaths must be traversed:

$$\lambda_1 = (1, p_1, 4, p_2, 6, 2)$$
$$\lambda_2 = (1, p'_1, 4, p'_2, 6, 7)$$

where p_1, p_2, p'_1, and p'_2 are definition clear subpaths with respect to variable z. In addition, because the last reference of variable z is inside a loop, the loop must be iterated a minimum and a larger number of times. The following test set covers C:

$$T = \left\{ \begin{array}{lllll} t_1: & < \mathtt{x} = -5 & \mathtt{y} = 2 & \mathtt{count} = 1 > \\ t_2: & < \mathtt{x} = -5 & \mathtt{y} = 2 & \mathtt{count} = 2 > \end{array} \right\}$$

The following path is traversed when Program P7.16 is executed against t_1:

$$(1, 2, 3, 4, 6, 7)$$

Clearly, the above path is of the form λ_2 where $p_1 = (2, 3)$ and p_2 is empty. Here the loop containing $u_6(z)$ is traversed once, i.e. minimum number of times.

The loop containing the following (complete) path is traversed when Program P7.16 is executed against t_2:

$(1, 2, 3, 4, 6, 2, 3, 6, 7)$

This path can be written as

$(1, p_1, 4, p_2, 6, 2, p_3)$

and also as

$(1, p'_1, 4, p'_2, 6, 7)$

where $p_1 = (2, 3)$, $p_2 = ()$, $p'_1 = (2, 3)$, and $p'_2 = (6, 2, 3)$. Note that p_2 is an empty subpath. p_1, p_2, p'_1, and p'_2 are all def-clear with respect to z. Note also the condition that subpaths λ_1 and λ_2 be traversed is satisfied. Thus, the test case $\{t\}$ covers the *k-dr* chain C.

7.4.5 Using the *k-dr* chain coverage

To determine the adequacy of a test set T for program P subject to requirements R, k is set to a suitable value. The larger the value of k, the more difficult it is to cover all the *k-dr* interactions. Given k, the next step is to determine all *l-dr* interactions for $1 \leq l \leq k$. We denote this set of all *l-dr* interactions by *k-dr*(k). The program is then executed against tests in T and the coverage determined. A formal definition of test adequacy with respect to *k-dr* coverage follows.

A test set is considered adequate with respect to the k-dr coverage criterion, for a given k, if all feasible k-dr chains are covered. Note that higher values of k may require more tests for coverage. However, at some point increasing the value of k will not lead to any new def-use chains unless one increases the number of times a loop body along a chain is executed.

k-dr *coverage*:
For a given $k \geq 2$, the k-dr*(k) coverage of T with respect to (P, R) is computed as:*

$$\frac{C_c^k}{(C^k - C_f^k)}$$

where C_c^k is the number of k-dr *interactions covered, C^k is the number of elements in* k-dr*(k), and C_f the number of infeasible interactions in* k-dr*(k). T is considered adequate with respect to the* k-dr*(k) coverage criterion if its* k-dr*(k) coverage is 1.*

7.4.6 Infeasible c- and p-uses

An infeasible path in a program may lead to infeasible c- or p-uses.

Coverage of a c- or a p-use requires a path to be traversed through the program. However, if this path is infeasible, then some c- and p-uses that require this path to be traversed might also be infeasible. Infeasible uses are often difficult to determine without some hint from a test tool. The next example illustrates why one c-use in Program P7.16 is infeasible.

Example 7.41 Consider the c-use at node 4 of z defined at node 5. For this c-use to be covered, control must first arrive at node 5 and then move to node 4 via a path that is def-clear with respect to z.

Further analysis reveals that for the control to first arrive at node 5, $x > 0$. For the control to then move to node 4, it must pass through nodes 6, 2, and 3. However, this is not possible because edge (2, 3) can be taken only if $x \leq 0$.

Note that x is not defined anywhere in the program except at node 1. Hence, if $x > 0$ when control first arrives at node 2, then it cannot be $x \leq 0$ in any subsequent iteration of the loop. This implies that the condition required for the coverage of c-use at node 4 of z defined at node 5 can never be true and hence this c-use is infeasible.

Example 7.42 A `for` loop that iterates for a fixed number of times can cause infeasible paths. Consider the following program that defines x prior to entering the loop and uses x inside the loop body and soon after loop exit. Thus there is a c-use of x in lines 3 and 7. A def-use graph for this program is shown in Figure 7.10.

Program P7.18

```
1   begin
2     int x,z,i
3     input (x); z=0;
4     for(i=1; i<5,  i++){
5       z=z+x*i;
6     }
7     output(x,z);
8   end
```

Path (`Start`, 1, 2, 4, `End`) must be traversed to cover the c-use of x at line 7. However, this path, and hence the c-use, is infeasible because the `for` loop body will be executed exactly five times and not zero times as required for this path to be traversed.

Figure 7.10 Data flow graph of P7.18.

7.4.7 Context coverage

Let P be the program under test, F its data flow graph, t a test case against which P is executed, $X(k)$ the set of input variables for node k in F, and $EDC(k)$ and $OEDC(k)$ be, respectively, the elementary data and ordered elementary contexts for node k. $EDC(k)$ is considered covered by t if the following two conditions are satisfied during the execution of P.

1. Node k is along the path traversed during the execution of P against t.
2. When control arrives at k along this path, all definitions in $EDC(k)$ are live.

A data context $DC(k)$ for node k is considered covered when all elementary data contexts in $DC(k)$ are covered. Similarly, an ordered elementary data context $OEDC(k)$ is considered covered by t if the following conditions are satisfied during the execution of P.

A test set is considered adequate with respect to the data context criterion if it covers all the feasible data contexts.

1. Node k is along the path p traversed during the execution of P against t,
2. All definitions in $OEDC(k)$ are live when control arrives at n along p.
3. The sequence in which variables in $X[k]$ are defined is the same as that in $OEDC(k)$.

An ordered data context $ODC(k)$ for node n is considered covered when all data contexts in $ODC(k)$ are covered.

Given a test set set T for program P subject to requirements R, formal definitions of test adequacy with respect to elementary data context and ordered elementary data context coverage follow.

Elementary data context coverage:
Given a program P subject to requirements R, and having a data flow graph containing n nodes, the ordered elementary data context coverage of T with respect to (P, R) is computed as:

$$\frac{EDC_c}{EDC - EDC_i}$$

where EDC is the number of elementary data contexts in P, EDC_c is the number of elementary data contexts covered, and EDC_i is the number of infeasible elementary data contexts. T is considered adequate with respect to the data context coverage criterion if the data context coverage is 1.

Ordered elementary data context coverage:
Given a program P subject to requirements R, and having a data flow graph containing n nodes, the ordered data context coverage of T with respect to (P, R) is computed as:

$$\frac{OEDC_c}{OEDC - OEDC_i}$$

where OEDC is the number of ordered elementary data contexts in P, $OEDC_c$ is the number of ordered elementary data conexts covered, and $OEDC_i$ is the number of infeasible ordered elementary data contexts. T is considered adequate with respect to the ordered elementary data context coverage criterion if the data context coverage is 1.

In the two definitions above, we have computed the coverage with respect to the elementary and ordered elementary data contexts, respectively. Another alternative is to compute the coverage with respect to data contexts and ordered data contexts (see Exercise 7.43). Such definitions would offer coarser coverage criteria than the ones above. For example, in such a coarse definition, a data context will be considered uncovered if any one of its constituent elementary data contexts is uncovered.

Example 7.43 Let us compute the adquacy of test T given below against the elementary data context coverage criteria.

$$T = \left\{ \begin{array}{lll} t_1: & <\mathtt{x}=-2 & \mathtt{y}=2 \quad \mathtt{count}=1> \\ t_2: & <\mathtt{x}=-2 & \mathtt{y}=-2 \quad \mathtt{count}=1> \\ t_3: & <\mathtt{x}=2 & \mathtt{y}=2 \quad \mathtt{count}=1> \\ t_4: & <\mathtt{x}=2 & \mathtt{y}=2 \quad \mathtt{count}=2> \end{array} \right\}$$

We assume that Program P7.16 is executed against the four test cases listed above. For each test case, the table below shows the path traversed, the elementary data context(s) covered, and the

cumulative elementary data context coverage (*EDC*). An asterisk against a node indicates that the corresponding elementary data context has already been covered and hence not counted in computing the cumulative *EDC*.

From Figure 7.7 and the table in Example 7.34, we obtain the total number of elementary data contexts as 12. Hence the cumulative *EDC* for test case t_j, $1 \leq j \leq 4$, is computed as the ratio $EDC_c/12$ where EDC_c is the total number of elementary data contexts covered after executing the program against t_j. Note that we could also compute data context coverage using the four data contexts for nodes 4 through 7 in Example 7.34. In that case, we will consider a data context as covered when all of its constituent elementary data contexts are covered.

Test case	**Path traversed**	**Elementary context (Node: context)**	**Cumulative *EDC* coverage**
t_1	(1, 2, 3, 4, 6, 7)	4: (d_1(y), d_1(z)) 6: (d_1(x), d_1(y), d_4(z), d_1(count)) 7: (d_4(z))	$\frac{3}{12} = 0.25$
t_2	(1, 2, 3, 6, 7)	6: (d_1(x), d_1(y), d_1(z), d_1(count)) 7: (d_1(z))	$\frac{5}{12} = 0.42$
t_3	(1, 2, 5, 6, 7)	5: (d_1(x)) 6: (d_1(x), d_1(y), d_5(z), d_1(count))	$\frac{8}{12} = 0.67$
t_4	(1, 2, 5, 6, 2, 5, 6, 7)	7 : (d_5(z)) 5*: (d_5(x)) 6*: (d_1(x), d_1(y), d_5(z), d_1(count)) 5*: (d_5(x)) 6 : (d_1(x), d_6(y), d_5(z), d_6(count)) 7*: (d_5(z))	$\frac{9}{12} = 0.85$

It is clear from the rightmost column in the table above that T is not adequate with respect to the elementary data context coverage criterion. Recall that T is adequate with respect to the MC/DC coverage criterion.

7.5 Control Flow Versus Data Flow

Adequacy criteria based on the flow of control aims at testing only a few of the many, sometimes infinitely large, paths through a program.

Data flow based adequacy criteria are generally stronger and hence more difficult to satisfy than control flow based adequacy criteria, including the MC/DC criterion.

For example, the block coverage criterion is satisfied when the paths traversed have touched each block in the program under test. Data flow based criteria has the same objective, i.e. these criteria assist in the selection of a few of the many paths through a program. For example, the c-use coverage criterion is satisfied when the paths traversed have covered all c-uses. However, sometimes data flow based criteria turn out to be more powerful than criteria based on the flow of control, including the MC/DC based criteria. As evidence in support of the previous statement, let us reconsider Program P7.16. Consider the following test case devised based on the program's requirements (which we have not listed explicitly).

$$T = \{t_1 : x = -2, y = 2, \texttt{count} = 2 >, t_2 : < x = 2, y = 2, \texttt{count} = 1 >\}$$

Surprisingly, test set $T = \{t_1, t_2\}$ is adequate for P7.16 with respect to all the control flow criteria described earlier except for the LCSAJ criterion. In terms of coverage numbers, block, condition, multiple condition, and MC/DC coverages are all 100%. However, the c-use coverage of T is only 58.3% and the p-use coverage is 75%. Translated in terms of error detection, the example illustrates that tests adequate with respect to data flow coverage criteria are more likely to expose program errors than those adequate with respect to criteria based exclusively on the flow of control.

Example 7.44 Suppose that line 14 in Program P7.16 has an error. The correct code at this line is:

```
14  y=x+y+z;
```

The following test set is adequate with respect to the MC/DC criterion.

$$T = \left\{ \begin{array}{llll} t_1: & < \mathrm{x} = -2 & \mathrm{y} = 2 & \texttt{count} = 1 > \\ t_2: & < \mathrm{x} = -2 & \mathrm{y} = -2 & \texttt{count} = 1 > \\ t_3: & < \mathrm{x} = 2 & \mathrm{y} = 2 & \texttt{count} = 1 > \\ t_4: & < \mathrm{x} = 2 & \mathrm{y} = 2 & \texttt{count} = 2 > \end{array} \right\}$$

Execution of P7.16 against T produces the following values of z:

P7.16 (t_1)=1.0
P7.16 (t_2)=0.0
P7.16 (t_3)=0.5
P7.16 (t_4)=0.5

It is easy to verify that the correct program also generates exactly the same values of z when executed against test cases in T. Thus T does not reveal

the simple error in P7.16. The c- and p-use coverage of T is, respectively, 0.75 and 0.5 (these numbers include the infeasible c-use). One of the c-uses that is not covered by T is that of y at line 8 corresponding to its definition at line 14. To cover this c-use, we need a test that ensures the execution of the following event sequence.

1. Control must arrive at line 14; this will happen for any test case.
2. Control must then get to line 8. For this to happen, `count` > 1, so that the loop does not terminate and conditions $x \leq 0$ and $y \geq 0$ must be true.

The following test case satisfies the conditions enumerated above.

$t_e : < x = -2, y = 2, \texttt{count} = 2 >$

Execution of Program P7.16 against t_e causes the c-use at line 8 to be covered. However, the value of z is 1.0 while the correct version of this program outputs 2.0 when executed against t_e thus revealing the error.

Though coverage of a c-use in the example above revealed the error, a key question one must ask is: "Will every test case that covers the c-use mentioned in Example 7.44 also reveal the error? This question is left as an exercise (see Exercise 7.38). It is also important to note that even though T in the above example does not reveal the error, some other MC/DC adequate test might.

7.6 The "Subsumes" Relation

We have discussed a variety of control flow and data flow based criteria to assess the adequacy of tests. These criteria assist in the selection of tests from a large, potentially infinite, set of inputs to the program under test. They also assist in the selection of a finite, and relatively small, number of paths through the program. All control and data flow based adequacy criteria are similar in that their objective is to assist a tester find a suitable subset of paths to be tested from the set of all paths in a given program. Given this similarity of objectives of the various adequacy criteria, the following questions are of interest.

A coverage criterion C_1 is considered to subsume coverage criterion C_2 if every test adequate with respect to C_1 is also adequate with respect to C_2.

Subsumes: Given a test set T that is adequate with respect to criterion C_1, what can we conclude about the adequacy of T with respect to another criterion C_2?

Effectiveness: Given a test set T that is adequate with respect to criterion C, what can we expect regarding its effectiveness in revealing errors?

In this chapter, we deal briefly with the first of the above two questions; an in-depth theoretical discussion of the subsumes relationship amongst various adequacy criteria is found in Volume 2. The effectiveness question is also dealt with in the research literature. The next example illustrates the meaning of the subsumes relationship.

Example 7.45 Program P7.19 takes three inputs x, y, and z, and computes p.

Program P7.19

```
1   begin
2     int x,y,z,p;
3     input (x,y,z);
4     if (y≤0) {
5      p=y*z+1;
6     }
7     else{
8       x=y*z+1;
9     }
10    if (z≤0) {
11     p=x*z+1;
12    }
13    else{
14     p=z*z+1;
15    }
16    output (p);
17   end
```

The following test set is adequate with respect to the p-use criterion for P7.19:

$$T = \left\{ \begin{array}{llll} t_1: & <\mathtt{x}=5 & \mathtt{y}=-2 & \mathtt{z}=1> \\ t_2: & <\mathtt{x}=5 & \mathtt{y}=2 & \mathtt{z}=-1> \end{array} \right\}$$

However, T does not cover the c-use $(d_3(\mathtt{x}), u_{11}(\mathtt{x}))$. This example illustrates that a test set that is adequate with respect to the p-use criterion is not necessarily adequate with respect to the c-use criterion.

Notice that for Program P7.19, a test that covers $(d_3(\mathtt{x}), u_{11}(\mathtt{x}))$ must cause the conditions at line 4 and 10 to evaluate to `true`. Such "coupling" of conditions is not required to obtain p-use adequacy (see Exercise 7.46). The next example illustrates that c-use adequacy does not imply p-use adequacy.

Example 7.46 Program P7.20 takes two inputs x and y, and computes z.

The subsumes relationship indicates the strength of an adequacy criterion. Generally, a stronger criterion is likely to be more effective in revealing program errors than a weaker criterion.

Figure 7.11 The subsumes relationship amongst various control and data flow based test adequacy criteria. $X \rightarrow Y$ indicates that X subsumes Y. See the bibliography section for citations to research that points to the assumptions that need to be satisfied for the relationship to hold.

While the all-uses criterion is empirically more powerful than the MC/DC criterion, it does subsume the MC/DC criterion.

Program P7.20

```
1   begin
2     int x,y,z=0,
3     input (x,y);
4     if (y≤0) {
5      z=y*x+1;
6     }
7      z=z*z+1;
8     output (z);
9   end
```

The following test is adequate with respect to the c-use criterion but not with respect to the p-use criterion.

$$T = \{\ t: \quad < x = 5 \quad y = -2 > \ \}$$

7.7 Structural and Functional Testing

Measuring code coverage while testing, and enhancing tests based on such measurements, is often referred to as "white box" or "structural" testing.

It is commonly believed that when testers measure code coverage, they are performing "structural testing" also known as "white-box" or "glass-box" testing. In addition, it is also said that while structural testing compares test program behavior against the apparent intention of the programmer as expressed in the source code, functional testing compares

test program behavior against a requirements specification. The difference between structural and functional testing is presented below in a different light.

As has been explained in Sections 7.1.2 and 7.1.3, measurement of code coverage is a way to assess the "goodness" or "completeness" of a test set T derived with the intention of checking whether or not the program P under test behaves in accordance with the requirements. Thus, when P is executed against a test case t that belongs to T, a tester indeed compares the behavior of P against its requirements as t itself is derived from the requirements.

Having successfully executed all tests in T, a tester might choose one of at least two options. One option is to assess the goodness of T using criteria not based on code coverage while the other is to use code coverage. If the tester chooses the first option, we say that the tester is performing functional testing *without the aid of any code based test adequacy measurement*. Certainly, the tester could use non-code based adequacy criteria, for example, requirements coverage.

However, when the tester chooses the second option, then some form of code coverage is measured and perhaps used to enhance T. If T is found adequate with respect to the code based adequacy criteria, say C, then the tester may decide to either further evaluate T using a more powerful adequacy criteria or be satisfied with T.

When T is not adequate with respect to C, the tester must continue to test P against the newly derived tests in the same way as P was tested against tests in T. For example, suppose that the coverage measurement indicates that decision d at, say, line 136 inside function `myfunc` has not been covered. We now expect the tester to determine, with respect to the requirements, why this decision was not covered. A few possible causes, using arbitrary labels for requirement, are given below.

- There was no test case in T to test for equirement $R_{39.3}$.
- Requirement R_4 was tested only with default parameters, testing it against any other value of an input parameter will cover d.
- Decision d can only be covered by a test that tests requirement $R_{2.3}$ followed by requirement $R_{1.7}$.

Once the cause for lack of coverage is determined, the tester proceeds to construct a test case t', not in T. P is then executed against t' to ensure that it behaves correctly in accordance with the requirements.

There are several possible outcomes of executing P against t'. One outcome is that decision d remains uncovered implying improper construction of t'. Another possibility is that P behaves incorrectly on t'. In either case,

testing is performed with respect to the requirements while the coverage measurement is used as a means to probe the inadequacy of T.

The discussion above leads us to conclude that structural testing is functional testing with the addition of code based adequacy measurement. Thus, rather than treat functional and structural testing as two independent test methods, we treat one as supplementing the other. In this case, structural testing is supplementary to functional testing. One performs structural testing as part of functional testing by measuring code coverage and using this measure to enhance tests generated using functional testing.

7.8 Scalability of Coverage Measurement

One might argue that measurement of code coverage, and test enhancement based on such measurement, is practical only during unit testing. One implication of such an argument is that it is impractical to perform code coverage measurements for integration or system tests. In the remainder of this section we discuss the practicality aspect of code coverage measurement. We offer suggestions on how testers could scale the coverage measurement and test enhancement process to large systems.

Measuring code coverage during unit tests is generally easy and can be done using tools such s JUnit. However, doing so for very large an complex applications is difficult for many reasons, for example, due to increase in time for execution, or the increased memory requirements.

Suppose that testers have access to the source code of the application under test. While it is possible to measure code coverage on binaries, lack of such access makes coverage measurement a difficult, if not an impossible, task. Several tools such as JBlanket developed by Joy Augustin, and Quilt available as Apache License, measure certain forms of code coverage by instrumenting the object code (byte code in Java). However, several other tools for coverage measurement often rely on instrumenting the source code and hence would fail to be of any use when the source is not available.

It is often easier to determine code coverage for non-embedded versus embedded systems.

Open systems: First, let us consider open systems, i.e. applications that are not embedded in any specific hardware and can be executed in open desktop environments. Examples of such applications abound and include products such as office tools, web services, and database systems.

Access to the code of any such application offers testers an opportunity to evaluate the tests using code based coverage criteria. However, the availability of a coverage measurement tool is almost a necessity when testing large systems. Tools for coverage measurement can be used to instrument the code. The coverage is monitored during execution and displayed after any number of executions. All coverage measurement tools display code coverage values in some form or another and can be used to determine test adequacy.

Incremental and selective coverage measurement: Coverage data for large applications can be overwhelming. For example, an application that contains 10,000 modules with a total of 45,000 conditions spread all over will lead to an enormous amount of coverage data. The display and storage of such data might create problems for the tester and the coverage measurement tool. For example, the test process might exhibit unacceptable slowdown, there might not be sufficient disk space to save all the coverage data, and the coverage measurement tool might breakdown as it might not have been tested to handle such large amounts of coverage data. Such problems can be overcome through incremental measurement and enhancement.

Even for large systems, coverage measurement can be made possible and effective through the use of an incremental approach.

Incremental coverage measurement is performed in several ways. One way is to create a hierarchy of code elements. Elements in the hierarchy are then prioritized using criteria such as to their frequency of use or their criticality to the acceptable operation of the application. Thus, in a large application with say 10,000 modules, one might create three sets of modules M_1, M_2, and M_3, with decreasing priority. The code measurement tool can then be asked to instrument and measure code coverage only in modules that belong to set M_1. Tests could then be enhanced based on the coverage data so obtained. The process could then be repeated for modules in M_2, and so on.

Reducing the number of modules for which coverage measurements are taken will reduce the burden on the coverage measurement tool, as well as the tester who must deal with the amount of coverage data and enhance the tests to improve coverage. However, such a reduction might not be sufficient. In that case, a tester could further subdivide elements of a module set, say M_1, into code elements and prioritize them. This further subdivision could be based on individual classes, or even methods. Such a subdivision further reduces the amount of code to be instrumented and the data to be analyzed by the tester for test enhancement.

In addition to constraining the portions of the code subject to coverage measurement, one could also constrain the coverage criteria. For example, one might begin with the measurement of method, or function, coverage in the selected modules. Once this coverage has been measured, tests enhanced, and adequate coverage obtained, one could move to the other portions of the code and repeat the process.

Embedded systems: Embedded systems pose a significant challenge in the measurement of code coverage. The challenge is primarily due to limited memory space and, in some cases, hard timing constraints. Instrumenting code while testing it in its embedded environment will certainly increase code size. The instrumented code might not fit in the available memory.

Even when it fits, it might affect the real-time constraints and impact the application behavior.

Memory constraints can be overcome in at least two ways. One way is to use the incremental coverage idea discussed above. Thus instrument only a limited portion of the code at a time and test. This would require the tests to be executed in several passes and hence increase the time to complete the coverage measurement.

Another approach to overcome memory constraints is hardware-based. One could use a tool that profiles program execution to obtain branch points in the machine code. Profiling is done in a non-intrusive manner by tapping the microprocessor-memory bus. These branch points are then mapped to the source program and coverage measures such as block and branch coverage computed and displayed.

Coverage measurement for object oriented programs: Object oriented (OO) programs allow a programmer to structure a program in terms of classes and methods. A class encapsulate one or more methods and serves as template for the creation of objects. Thus, while all code coverage criteria discussed in this chapter applies to OO programs, some additional coverage measures can also be devised. These include, method coverage, context coverage, object coverage, and others.

7.9 Tools

A number of open source and commercially available tools exist for measuring code coverage for a given test suite. Below is a sample of such tools.

`Tool`: PaRTeG
`Link`: `http://parteg.sourceforge.net/`
`Description`

This is a tool that fits several chapters in this book. It generate tests from UML state machines and class diagrams. Test sets generated from code satisfy MC/DC, state coverage, transition coverage, decision coverage, multiple condition coverage, and boundary coverage. It is also able to generate mutants for Java classes. PaRTeG is available for download as an Eclipse plugin.

`Tool`: Cobertura
`Link`: `http://cobertura.sourceforge.net/`
`Description`

This is a freely available tool for measuring code coverage of Java code. It instruments byte code, not Java source, to measure coverage. Statement and branch coverages are measured. Measurement statistics, e.g., number of times a statement has been exercised, are displayed in a graphical form. Lines and branches not covered are also mapped to source code and highlighted.

`Tool`: EclEmma
`Link`: `http://cobertura.sourceforge.net/`
`Description`

This is another tool for measuring coverage of Java programs. It is available as an Eclipse as well as JUnit plugin. Statement coverage is supported. Coverage is displayed graphically for each method.

`Tool`: Bullseye
`Link`: `http://www.bullseye.com/`
`Description`

This is a commercial tool for measuring coverage of C++ programs. It instruments the source code. It is able to measure function coverage, condition coverage, and decision coverage.

`Link`: `http://www.microsoft.com/visualstudio/en-us`
`Tool`: Visual Studio
`Description`

This is an integrated development environment for .Net and C# programs. It measures statement, block and condition coverage.

`Tool`: LDRA Testbed

A summary of this tool appears in Chapter 2.

SUMMARY

This chapter covers the foundations of test assessment and enhancement using structural coverage criteria. We have divided these criteria into control flow based and data flow based adequacy criteria. Each criterion is defined and illustrated with examples. Examples also show how faults are detected, or missed, when a criterion is satisfied. The simplest of all control flow based criteria such as statement coverage and decision coverage have found their way in to a large number of commercially available testing tools.

We have introduced some of the several data flow based adequacy criteria, namely the all-uses, p-uses, and the c-uses. There are others not covered here and referred to in the bibliographic notes section. While adequacy with respect to data flow based criteria is stronger than that based on control flow based criteria, it has not found much use in commercial software development. However, the availability of good tools such as χSUDS from Telcordia Technologies, and education, promises to change this state of affairs.

This chapter is certainly not a complete treatment of all the different control and data flow based adequacy criteria. There are many more, e.g. data context coverage, and others. Some of the uncovered material is pointed to in the section on bibliographic notes.

Exercises

7.1 Consider the following often found statement in research publications: "Complete testing is not possible." What completeness criterion forms the basis of this statement? In light of the completeness criteria discussed in this chapter, is this statement correct?

7.2 Enumerate all paths in P7.5. Assume that each loop is required to be traversed zero times and once.

7.3 Why did we decide to input all tests in T during a single execution of P7.5? Will the error be revealed if we were to input each test in a separate execution?

7.4 Draw the control flow graph for P7.5. Enumerate all paths through this program that traverse the loop zero times and once.

7.5 Let A, B, and C denote three Booleans. Let $Cond = (A$ `and` $B)$ `or` $(A$ `and` $C)$ `or` $(B$ `and` $C)$ be a condition. What is the minimum number of tests required to (a) cover a decision based on *Cond* and (b) cover all atomic conditions in *Cond*?

7.6 Construct a program in your favorite language that inputs three integers x, y, and z and computes output O using the specification in Table 7.12. Assume that $f_1(x, y, z) = x + y + z$ and $f_2(x, y, z) = x * y * z$.

Introduce an error in your program. Now construct a test set T that is decision-adequate, but is not multiple condition-adequate, and does *not* reveal the error you introduced. Enhance T so that it is multiple condition-adequate and *does* reveal the error. Will all tests that are multiple condition-adequate for your program reveal the error?

7.7 Suppose that test set T for a given program and the corresponding requirements is adequate with respect to the statement coverage criterion. Will T be always adequate with respect to the block coverage criterion?

Table 7.12 Computing O for the program in Exercise 7.6.

	$x < y$	$x < z$	O
1	`true`	`true`	$f_1(x, y, z)$
2	`true`	`false`	$f_2(x, y, z)$
3	`false`	`true`	$f_2(x, y, z)$
4	`false`	`false`	$f_1(x, y, z)$

7.8 (a) How many decisions are introduced by an `if` statement? (b) By a `switch` statement? (c) Does each decision in a program lead to a new path?

7.9 Consider the following claim: *If the number of simple conditions in a decision is n, then the number of tests required to cover all conditions is* 2^n. If possible, construct a program to show that in general this claim is not true.

7.10 Let A, B, and C denote three Booleans. Let $C' = (A$ `and` $B)$ `or` $(A$ `and` $C)$ `or` $(B$ `and` $C)$ be a condition. What is the minimum number of tests required to (a) cover a decision based on C' and (b) cover all conditions in C'?

7.11 Modify T in Example 7.15 to render it adequate with respect to the condition/decision criterion for P7.9.

7.12 Given simple conditions A, B, and C, derive minimal MC/DC adequate tests for the following conditions: (a) A `and` B `and` C, (b) A `or` B `or` C, and (c) A `xor` B. Is the test set unique in each case?

7.13 While removing redundant tests from Table 7.6, we selected tests (3, 4) to cover condition C_3. What will be the minimal MC-adequate test set if, instead, tests (5, 6) or (7, 8) are selected?

7.14 Are the tests, given for each of the three conditions in (a) Table 7.9 and (b) Table 7.10, unique?

7.15 In Section 7.2.9 we list a simple procedure to derive MC/DC adequate tests for condition $C = C_1$ `and` C_2 `and` C_3 from MC/DC adequate tests for condition $C = C_1$ `and` C_2. Using this procedure as a guide, construct another procedure to (a) derive MC/DC adequate tests for $C = C_1$ `or` C_2 `or` C_3, given MC/DC adequate tests for $C = C_1$ `or` C_2 and (b) derive MC/DC adequate tests for $C = C_1$ `xor` C_2 `xor` C_3, given MC/DC adequate tests for $C = C_1$ `xor` C_2.

7.16 Extend the procedure in Section 7.2.9 to derive MC/DC adequate tests for $C = C_1$ `and` $C_2 \ldots$ `and` C_{n-1} `and` C_n, given MC/DC adequate tests for $C = C_1$ `and` $C_2 \ldots$ `and` C_{n-1}.

7.17 Use the procedures derived in Exercises 7.15 and 7.16 to construct an MC/DC adequate test for (a) $(C_1$ `and` $C_2)$ `or` C_3, (b) $(C_1$ `or` $C_2)$ `and` C_3, and (c) $(C_1$ `or` $C_2)$ `xor` C_3.

7.18 Suppose test T is found adequate with respect to the condition coverage for some program P. Is it also adequate with respect to decision coverage? Is the opposite true?

7.19 Consider program P containing exactly one compound condition C and no other conditions of any kind. C comprises n simple conditions. Let T be the test set adequate for P with respect to the MC/DC criterion. Show that T contains at least n and at most $2n$ test cases.

7.20 Consider a test set T adequate for program P with respect to some coverage criterion C. Now suppose that $T_1 \cup T_2 = T$ and that both T_1 and T_2 are adequate with respect to C. Let P' be a modified version of P. To test P', under what conditions would you execute P' against all of T? Only T_1? Only T_2?

7.21 Is T_3 of Example 7.23 minimal with respect to MC/DC coverage? If not then remove all redundant test cases from T_3 and generate a minimal test set.

7.22 (a) In Example 7.23, suppose that condition C_2 at line 12 is incorrectly coded as:

```
12    if((z * z > y) and (prev==“East”))
```

Will the execution of P7.14 against t_5 reveal the error?
(b) Now suppose that in Example 7.23, condition C_3 at line 14 is also incorrectly coded as:

```
12    else if (z * z ≤ y) or (current==“South”))
```

Note that there are two erroneous statements in the program. Will the execution of the erroneous P7.14 against t_5 reveal the error?
(c) Construct a test case that will cause the erroneous P7.14 to produce an incorrect output thereby revealing the error.
(d) What errors in the requirements will likely be revealed by each of the tests $t_5, t_6, \ldots, t_9$ in Example 7.23?

7.23 Why does the MC/DC adequate test set in Example 7.24 contain four tests when MC/DC adequacy for a conjunction of two simple conditions can be achieved with only three tests?

7.24 (a) Given $C = C_1$ `and` C_2 `and` ... `and` C_n, suppose that C has been coded as $C' = C_1$ `and` C_2 `and` ... `and` C_{i-1} `and` C_{i+1} ... `and` C_n; note one missing simple condition in C'. Show that, in the general case, an MC/DC adequate test set is not *likely* to detect the error in C'. (b) Can we say that an MC/DC adequate test set *will not* detect this error? (c) Does the error detection capability of the MC/DC criteria change as the number of missing simple conditions increases? (d) Answer parts (a) and (c) assuming that $C = C_1$ `or` C_2 `or` ... `or` C_n and $C' = C_1$ `or` C_2 `or` ... `or` C_{i-1} `or` C_{i+1} ... `or` C_n.

7.25 (a) Suppose that P7.14 contains one error in the coding of condition C_2. The error is a missing simple condition. The incorrect condition is:

```
12    if((z * z > y) and (prev==“East”))
```

Develop an MC/DC adequate test set for the incorrect version of P7.14. How many of the test cases derived reveal the error? Does the answer to this question remain unchanged for every minimal MC/DC adequate test for this incorrect program?
(b) Answer (a) for the following error in coding C_3:

```
14    else if ((x<y) or (z * z ≤ y) and (current==“South”))
```

7.26 Let A, B, and C denote Boolean expressions. Consider a compound condition D defined as follows:

$D = (A$ `and` $B)$ `or` $(A$ `and` $C)$ `or` $(B$ `and` $C)$

Assuming that D is used in a decision, construct a minimal set of tests for D that are adequate with respect to (a) decision coverage, (b) condition coverage, (c) multiple condition coverage, and (d) MC/DC coverage.

7.27 Show that a test set adequate with respect to the LCSAJ(MC/DC) criterion may not be adequate with respect to the LCSAJ (MC/DC) criterion.

7.28 There are at least two options to choose from when considering data flow coverage of arrays. One option is to treat the entire array as a single variable. Thus whenever this variable name appears in the context of an assignment, the entire array is considered defined; when it appears in the context of a use, the entire array is considered used. The other option is to treat each element of the array as a distinct variable.
Discuss the impact of the two options on (a) the cost of deriving data flow adequate tests and (b) the cost of keeping track of the coverage values in an automated data flow coverage tool.

7.29 (a) Prove that basis path coverage implies decision coverage. (b) Does decision coverage imply basis path coverage?

7.30 Consider the following statement: "If a graph contains a loop, it has an infinite number of paths." Is this statement correct?

7.31 Paul Amman and Jeff Offutt have defined simple and prime paths as follows. A *simple path* from node p to node q in a CFG is a subpath in which no node appears more than once, except that the first and the last nodes might possibly be p. A *prime path* is a simple path that does not appear as a subpath of any other simple path. A *round trip path* is a prime path that starts and ends at the same node.
(a) Show that coverage of all prime paths implies coverage of all decisions.
(b) Show that prime path coverage does not imply MC/DC coverage unless each condition in the program is simple.
(c) Show that prime path coverage implies def-use coverage when all path conditions are simple.
(d) Construct all simple, prime, and round trip paths for the CFG in Figure 2.5(b) and (c).

7.32 (a) In Example 7.30, $(1, 2, 5, 6)$ is considered a def-clear path with respect to `count` defined at node 1 and used at node 5. Considering that `count` is defined at node 6, why is this redefinition of `count` not masking the definition of `count` at node 1? (b) Construct two complete paths through the data flow graph in Figure7.7 that are def-clear with respect to the definition of `count` at node 6 and its use in the same node.

7.33 In Example 7.37, suppose that a test set T contains only t_p. Is T adequate with respect to c-use coverage? p-use coverage? Generate additional tests to obtain complete c- and p-use coverages if T is inadequate with respect to these criteria.

7.34 In Section 7.3.8 a minimal set of c- and p- uses is listed that when covered by a test set T ensure that all feasible c- and p-uses are also covered by T. Show that the given set is minimal.

7.35 (a) Show that indeed set $T = \{t_c, t_p\}$, consisting of tests derived in Examples 7.36 and 7.37, is adequate with respect to the p-use criterion. (b) Enhance T to obtain T' adequate with respect to the c-use criterion. (c) Is T' minimal, i.e. can you reduce T' and obtain complete c- and p-use coverage from the reduced set?

7.36 Consider a path s through some program P. Variable `v` is defined along this path at some node n_d and used subsequently at some node n_u in an assignment, i.e. there is a c-use of `v` along s. Suppose now that path s is infeasible. Does this imply that the c-use of `v` at node n_u is also infeasible? (b) Suppose now that there is a p-use of `v` at node n_p along path s. Given that s is infeasible, is this p-use also infeasible? Explain your answer.

7.37 Consider the following program, due to Boyer, Elapas, and Levitt, and analyzed by Rapps and Weyuker in the context of data flow testing in their seminal publication. The program finds the square root of input `p`, $0 \leq$ `p` < 1 to accuracy `e`, $0 <$ `e` ≤ 1.

Program P7.21

```
1   begin
2      float x=0, p, e, d=1, c;
3      input (p, e);
4      c=2*p;
5      if (c≥2) {
6        output ("Error");
7      }
8      else{
9       while (d>e) {
10           d=d/2; t=c-(2*x+d);
11           if (t≥0) {
12             x=x+d;
13             c=2*(c-(2*x+d));
14           }
15           else{
16           c=2*c;
17           }
18         }
19         output ("Square root of p='', x);
20      }
21   end
```

For P7.21, (a) draw a data flow graph, (b) develop an MC/DC adequate test set T_{mcdc}, and (c) enhance T_{mcdc} to generate T that is adequate with respect to the all-uses criterion. (d) There is an error in P7.37. Lines 12 and 13 must be interchanged to remove this error. Does T reveal the error? (e) Does T_{mcdc} reveal the error? (f) Will every T, adequate with respect to the all-uses criterion, reveal the error?

7.38 (a) In Example 7.44, what condition must be satisfied by a test case for Program P7.16 to produce an incorrect output? Let us refer to the condition you derive as C_1. (b) Is C_1 the same

as the condition, say C_2, required to cover the c-use of y at line 8 corresponding to its definition at line 14? (c) For Program P7.21, what would you conclude regarding the detection if C_1 does not imply C_2, or vice versa?

7.39 (a) While doing *k-dr* analysis, it is assumed that each node in the data flow graph corresponds to a simple predicate. Modify the data flow graph shown in Figure 7.6 so that each node in the modified graph corresponds to a simple predicate. (b) Does *k-dr* coverage for $k = 2$ imply condition coverage? Multiple condition coverage? MC/DC coverage?

7.40 (a) Draw a data flow graph F for Program P7.22. (b) Construct data context sets for all nodes in F. (c) Construct a test set adequate with respect to data context coverage. Assume that functions foo1(), foo2(), foo3(), foo4(), P1(), and P2() are defined elsewhere.

Program P7.22

```
1   begin
2     int x,y,z;
3     input (x,y);
4     y= foo1(x,y);
5     while (P1(x)){
6       z= foo2(x,y);
7       if (P2(x,y) {
8         y= foo3(y);
9       }
10      else {
11        x= foo4(x);
12      }
13    }        // End of loop.
14    output (z);
15  end
```

7.41 (a) The table in Example 7.34 lists only the feasible elementary data contexts for various nodes for Program P7.16. Enumerate all elementary data contexts for nodes 4, 5, 6, and 7. Amongst these, identify the infeasible elementary data contexts. (b) How will the answer to (a) change if the initialization $z = 0$ is removed and the statement at line 4 is replaced by:

4 input (x, y, z, count);

7.42 Construct $ODC(n)$ for all nodes in the data flow graph for Program P7.22. Is the test set constructed in Exercise 7.40 adequate with respect to the ordered data context coverage criterion? If not then enhance it to a set that is adequate with respect to the ordered data context coverage criterion.

7.43 (a) Define data context coverage and ordered data context coverage in a manner analogous to the definitions of elementary and ordered elementary data contexts, respectively. (b) Which of the two sets of definitions of context coverage would you recommend, and why, for the measurement of test adequacy and enhancement?

7.44 Find all infeasible elementary data contexts for node 6 in Figure 7.7.

7.45 For Program P7.23, construct a test set adequate with respect to the p-use criterion but not with respect to the c-use criterion.

Program P7.23

```
1   begin
2      int x,y,c, count;
3      input (x,count);
4      y=0;
5      c=count;
6      while (c>0) {
7       y=y*x;
8       c=c-1;
9      }// End of loop
10      output (y);
11   end
```

7.46 Show that for programs containing only one conditional statement, and no loops, the p-use criterion subsumes the c-use criterion.

7.47 The statement "Structural testing compares test program behavior against the apparent intention of the source code." can also be restated as "Functional testing compares test program behavior against the apparent intention of the source code." Do you agree with these statements? Explain your answer.

7.48 Solve parts (a) and (b) of Exercise 7.28 in Chapter 8.

7.49 Boundary-interior testing proposed by William Howden requires a loop to be iterated zero times and at least once. Construct a program with an error in the loop that can be revealed only if the loop is iterated at least twice. You may use nested loops. Note that it is easy to construct such an example by placing the construction of a function f inside the loop and executing f only in the second or subsequent iteration. Try constructing a more sophisticated program.

7.50 Suppose that a program contains two loops, one nested inside the other. What is the minimum number of boundary-interior tests required to satisfy Howden's boundary-interior adequacy criterion?

7.51 Critique the following statement obtained from a web site: "White Box tests suffer from the following shortcomings: (a) Creating a false sense of stability and quality. An application can often pass all unit tests and still appear unavailable to users due to a non-functioning component (a database, connection pool, etc.). This is a critical problem, since it makes the IT application team look incompetent, (b) Inability to validate use-case completeness because they use a different interface to the application."

8

Test Adequacy Assessment using Program Mutation

CONTENTS

8.1 Introduction
8.2 Mutation and mutants
8.3 Test assessment using mutation
8.4 Mutation operators
8.5 Design of mutation operators
8.6 Founding principles of mutation testing
8.7 Equivalent mutants
8.8 Fault detection using mutation
8.9 Types of mutants
8.10 Mutation operators for C
8.11 Mutation operators for java
8.12 Comparison of mutation operators
8.13 Mutation testing within budget
8.14 CASE and program testing
8.15 Tools

The purpose of this chapter is to introduce Program Mutation as a technique for the assessment of test adequacy and the enhancement of test sets. The chapter also focuses on the design and use of mutation operators for procedural and objected-oriented programming languages.

8.1 Introduction

Program mutation is a powerful technique for the assessment of the goodness of tests. It provides a set of strong criteria for test assessment and enhancement. If your tests are adequate with respect to some other adequacy criterion, such as the modified condition/decision coverage (MC/DC) criterion, then chances are that these are not adequate with respect to most criteria offered by program mutation.

Program mutation is a technique for assessing the adequacy (goodness) of test sets and for enhancing inadequate tests.

Program mutation is a technique to assess and enhance your tests. Hereafter we will refer to "program mutation" as "mutation." When testers use mutation to assess the adequacy of their tests, and possibly enhance them, we say that they are using *mutation testing*. Sometimes the act of assessing test adequacy using mutation is also referred to as *mutation analysis*.

Given that mutation requires access to all or portions of the source code, it is considered as a white-box, or code-based, technique. Some refer to mutation testing as fault injection testing. However, it must be noted that fault-injection testing is a separate area in its own right and must be distinguished from mutation testing as a technique for test adequacy and enhancement.

Program mutation can be used as a white-box as well as a black-box technique.

Mutation has also been used as a black-box technique. In this case, it is used to mutate specifications and, in the case of web applications, messages that flow between a client and a server. This chapter focuses on the mutation of computer programs written in high-level languages such as Fortran, C, and Java.

While mutation of computer programs requires access to the source code of the application under test, some variants can do without. Interface mutation requires access only to the interface of the application under test. Run-time fault injection, a technique similar to mutation, requires only the binary version of the application under test.

Mutation can be used to assess and enhance tests for program units, such as C functions and Java classes. It can also be used to assess and enhance tests for an integrated set of components. Thus, as explained in the remainder of this chapter, mutation is a powerful technique for use during unit, integration, and system testing.

Program mutation can be applied to assess tests for small program units, e.g., C functions or Java classes, or complete applications.

A cautionary note: Mutation is a significantly different way of assessing test adequacy than what we have discussed in the previous chapters. Thus, while reading this chapter, you will likely have questions of the kind "Why this . . . " or "Why that . . . " With patience, you will find that most, if not all, of your questions are answered in this chapter.

A mutant of program *P* is obtained by making a slight change to *P*. Thus, a program and its mutant are syntactically almost the same though could differ significantly in terms of their behaviors.

A program *P* under test is mutated by applying a mutation operator to P that results in a mutant.

8.2 Mutation and Mutants

Mutation is the act of changing a program, albeit only slightly. If *P* denotes the original program under test and *M* a program obtained by slightly changing *P*, then *M* is known as a *mutant* of *P* and *P* the *parent* of *M*. Given that *P* is syntactically correct, and hence compiles, *M* must be syntactically correct. *M* might or might not exhibit the behavior of *P* from which it is derived.

The term "to mutate" refers to the act of mutation. To "mutate" a program means to change it. Of course, for the purpose of test assessment, we mutate by introducing only "slight" changes.

Example 8.1 Consider the following simple program.

Program P8.1

```
begin
  int x, y;
  input (x, y);
  if (x<y)
     output (x+y);
  else
     output (x*y);
end
```

A large variety of changes can be made to P8.1 such that the resultant program is syntactically correct. Below we list two mutants of P8.1. Mutant M1 is obtained by replacing the $<$ operator by the $\leq$ operator. Mutant M2 is obtained by replacing the $*$ operator by the / operator.

Mutant M1 of Program P8.1

```
begin
  int x, y;
  input (x, y);
  if (x≤y)          ← Mutated statement
  then
    output (x+y);
  else
    output (x*y);
end
```

Mutant M2 of Program P8.1

```
begin
  int x, y;
  input (x, y);
```

```
4    if(x<y)
5    then
6      output (x+y);
7    else
8      output (x/y);          ← Mutated statement
9   end
```

Notice that the changes made in the original program are simple. For example, we have not added any chunk of code to the original program to generate a mutant. Also, only one change has been made to the parent to obtain its mutant.

8.2.1 First order and higher order mutants

Mutants generated by introducing only a single change to a program under test are also known as *first order* mutants. Second order mutants are created by making two simple changes, third order by making three simple changes, and so on. One can generate a second order mutant by creating a first order mutant of another first order mutant. Similarly, an nth order mutant can be created by creating a first order mutant of an $(n-1)^{th}$ order mutant.

A first order mutant of program P is obtained by making a single change to P using a single mutation operator once. Applying a mutation operator more than once or applying multiple mutation operators simultaneously to P leads to a higher order mutant.

Example 8.2 Once again let us consider Program P8.1. We can obtain a second order mutant of this program in a variety of ways. Here is a second order mutant obtained by replacing variable y in the `if` statement by the expression $y+1$, and replacing operator $+$ in the expression $x+y$ by the operator $/$.

Mutant M3 of Program P8.1

```
1   begin
2     int x, y;
3     input (x, y);
4     if(x<y+1)          ← Mutated statement
5     then
6       output (x/y);    ← Mutated statement
7     else
8       output (x*y);
9   end
```

Mutants, other than first order, are also known as *higher order* mutants. First order mutants are the ones generally used in practice. There are two reasons why first order mutants are preferred to higher order mutants. One

For a given program P the number of higher order mutants could be significantly larger than the number of first order mutants.

reason is that there are many more higher order mutants of a program than first order mutants. For example, 528,906 second order mutants are generated for program FIND that contains only 28 lines of Fortran code. Such a large number of mutants create a scalability problem during adequacy assessment. Another reason has to do with the coupling effect and is explained in Section 8.6.2.

Note that so far we have used the term "simple change" without explaining what we mean by "simple" and what is a "complex" change. An answer to this question appears in the following sections.

8.2.2 Syntax and semantics of mutants

Syntax is the carrier of semantics. Thus, making even a simple syntactic change in a program might lead to a drastic change in the program's run time behavior, or none.

In the examples presented so far, we mutated a program by making simple syntactic changes. Can we mutate using semantic changes? Yes, we certainly can. However, note that syntax is the carrier of semantics in computer programs. Thus, given a program P written in a well-defined programming language, a "semantic change" in P is made by making one or more syntactic changes. A few illustrative examples follow.

Example 8.3 Let $f_{P8.1}(x, y)$ denote the function computed by Program P8.1. We can write $f(x, y)$ as follows:

$$f_{P8.1}(x, y) = \begin{cases} x + y & \text{if } x < y \\ x * y & \text{otherwise} \end{cases}$$

Let $f_{M1}(x, y)$ and $f_{M2}(x, y)$ denote the functions computed by M1 and M2, respectively. We can write $f_{M1}(x, y)$ and $f_{M2}(x, y)$ as follows:

$$f_{M1}(x, y) = \begin{cases} x + y & \text{if } x \leq y \\ x * y & \text{otherwise} \end{cases}$$

$$f_{M2}(x, y) = \begin{cases} x + y & \text{if } x < y \\ x / y & \text{otherwise} \end{cases}$$

Notice that the three functions $f_{P8.1}(x, y)$, $f_{M1}(x, y)$, and $f_{M2}(x, y)$ are different. Thus, we have changed the semantics of P8.1 by changing its syntax.

The previous example illustrates the meaning of "syntax is the carrier of semantics." Mutation might, at first thought, seem to be "just" a simple syntactic change made to a program. In effect, such a simple syntactic change could have a drastic effect, or it might have no effect at all, on program semantics. The next two examples illustrate why.

Example 8.4 Nuclear reactors are increasingly relying on the use of software for control. Despite the intensive use of safety mechanisms, such as the use of 400,000 liters of cool heavy water moderator, the control software must continually monitor various reactor parameters and respond appropriately to conditions that could lead to a meltdown. For example, the Darlington Nuclear Generating Station located near Toronto, Canada, uses two independent computer–based shutdown systems. Though formal methods can be, and have been, used to convince the regulatory organizations that the software is reliable, a thorough testing of such systems is inevitable.

Thorough testing of critical software systems is inevitable even after the software has been formally verified.

While the decision logic in a software-based emergency shutdown system would be quite complex, the highly simplified procedure below indicates that simple changes to a control program might lead to disastrous consequences. Assume that the *checkTemp* procedure is invoked by the reactor monitoring system with three most recent sensory readings of the reactor temperature. The procedure returns a danger signal to the caller through variable `danger`.

Program P8.2

```
enum dangerLevel {none, moderate, high, veryHigh};
procedure checkTemp (currentTemp, maxTemp){
  float currentTemp[3], maxTemp; int  highCount=0;
  enum dangerLevel danger;
  danger=none;
  if (currentTemp[0]>maxTemp)
   highCount=1;
  if (currentTemp[1]>maxTemp)
   highCount=highCount+1;
  if (currentTemp[2])>maxTemp)
   highCount=highCount+1;
  if (highCount==1) danger=moderate;
  if (highCount==2) danger=high;
  if (highCount==3) danger=veryHigh;
  return(danger);
}
```

Procedure `checkTemp` compares each of the three temperature readings against the maximum allowed. A “none” signal is returned if none of the three readings is above the maximum allowed. Otherwise, a “moderate,” “high,” or “veryHigh” signal is returned depending on, respectively, whether one, two, or three readings are above the allowable maximum. Now consider the following

mutant of P8.2 obtained by replacing the constant "veryHigh" with another constant "none" at line 14.

Mutant M1 of Program P8.2

```
1   enum dangerLevel {none, moderate, high, veryHigh};
2   procedure checkTemp (currentTemp, maxTemp){
3    float currentTemp[3], maxTemp; int highCount=0;
4    enum dangerLevel danger;
5    danger=none;
6    if (currentTemp[0]>maxTemp)
7     highCount=1;
8    if (currentTemp[1]>maxTemp)
9     highCount=highCount+1;
10   if (currentTemp[2])>maxTemp)
11    highCount=highCount+1;
12   if (highCount==1) danger=moderate;
13   if (highCount==2) danger=high;
14   if (highCount==3) danger=none; ← Mutated statement
15   return(danger);
16  }
```

A simple syntactic change in a critical software system may lead to a disastrous change in its run time behavior.

Notice the difference in the behaviors of P8.2 and M1. Program P8.2 returns "veryHigh" signal when all three temperature readings are higher than the maximum allowable, its mutant M1 returns the "none" signal under the same circumstances.

While the syntactic change made to the program is trivial, the resultant change in its behavior could lead to an incorrect operation of the reactor shutdown software potentially causing a major environmental and human disaster. It is such changes in behavior that we refer to as *drastic*. Indeed, the term "drastic" in the context of software testing often refers to the nature of the possible consequence of a program's, or its mutant's, behavior.

Example 8.5 While a simple change in a program might create a mutant whose behavior differs drastically from its parent, another mutant might behave exactly the same as the original program. Let us examine the following mutant obtained by replacing the == operator at line 12 in P8.2 by the ≥ operator.

Mutant M2 of Program P8.2

```
1   enum dangerLevel {none, moderate, high, veryHigh};
2   procedure checkTemp (currentTemp, maxTemp){
3    float currentTemp[3], maxTemp; int highCount=0;
```

```
  enum dangerLevel danger;
  danger=none;
  if (currentTemp[0]>maxTemp)
    highCount=1;
  if (currentTemp[1]>maxTemp)
    highCount=highCount+1;
  if (currentTemp[2]) >maxTemp)
    highCount=highCount+1;
  if (highCount≥1) danger=moderate; ← Mutated statement
  if (highCount==2) danger=high;
  if (highCount==3) danger=veryHigh;
  return(danger);
}
```

It is easy to check that for all triples of input temperature values and the maximum allowable temperature, P8.2 and M2 will return the same value of `danger`. Certainly, during execution, M2 will exhibit a different behavior than that of P8.2. However, the values returned by both the program and its mutant will be identical.

The behavior of a program P and its mutant M can be compared at different points during execution. In most practical applications of mutation testing the behavior compared by an examination the observable (external) outputs of P and M during or at the end of execution.

As shown in the examples above, while a mutant might behave differently than its parent during execution, in mutation testing the behavior of the two entities are considered identical if they exhibit identical behaviors at the specified points of observation. Thus, when comparing the behavior of a mutant with its parent, one must be clear about what are the points of observation. This leads us to strong and weak mutations.

8.2.3 Strong and weak mutations

We mentioned earlier that a mutant is considered distinguished from its parent when the two behave differently. A natural question to ask is "At what point during program execution should the behaviors be observed?" We deal with two kinds of observations *external* and *internal*. Suppose we decide to observe the behavior of a procedure soon after its termination. In this case we observe the return value, and any side effects in terms of changes to values of global variables and data files. We shall refer to this mode of observation as *external*.

An *internal* observation refers to the observation of the state of a program, and its mutant, during their respective executions. Internal observations could be done in a variety of ways that differ in where the program state is observed. One could observe at each state change, or at specific points in the program.

In string mutation only the external outputs of program P and its mutant M are compared, to determine whether or not the two have differently. In weak mutation the comparison could be carried out at intermediate stages during the execution and among internal state variables of P and M.

Strong mutation testing uses external observation. Thus, a mutant and its parent are allowed to run to completion at which point their respective outputs are compared. Weak mutation testing uses internal observation. It is possible that a mutant behaves similar to its parent under strong mutation but not under weak mutation.

Example 8.6 Suppose we decide to compare the behaviors of Program P8.2 and its mutant M2 using external observation. As mentioned in the previous example, we will notice no difference in the behaviors of the two programs as both return the same value of variable `danger` for all inputs.

Instead, suppose we decide to use internal observation to compare the behaviors of Program P8.2 and M2. Further, the state of each program is observed at the end of the execution of each source line. The states of the two programs will differ when observed soon after line 12 for the following input t:

$$t : < \texttt{maxTemp} = 1200, \texttt{currentTemp} = [1250, 1389, 1127] >$$

For the inputs above, the states of P8.2 and M2 observed soon after the execution of the statement at line 12 are as follows:

	`danger`	`highCount`
P8.2	none	2
M2	moderate	2

A program and its mutant might be equivalent under strong mutation but not under weak mutation.

Note that while the inputs are also a part of a program's state, we have excluded them from the table above as they do not change during the execution of the program and its mutant. The two states shown above are clearly different. However, in this example, the difference observed in the states of the program and its mutant does not effect their respective return values, which remain identical.

For all inputs, M2 behaves identically to its parent P8.2 under external observation. Hence, M2 is considered equivalent to P8.2 under strong mutation. However, as revealed in the table above, M2 is distinguishable from P8.2 by test case t. Thus, the two are not equivalent under weak mutation.

8.2.4 Why mutate?

It is natural to wonder why a program should be mutated. Before we answer this question, let us consider the following scenario likely to arise in practice.

Suppose you have developed a program, say, P. You have tested P. You found quite a few faults in P during testing. You have fixed all these faults and have retested P. You have even made sure that your test set is adequate with respect to the MC/DC criterion. After all this tremendous effort you have put into testing P, you are now justifiably confident that P is correct with respect to its requirements.

Happy and confident, you are about to release P to another group within your company. At this juncture in the project, someone takes a look at your program and, pointing to a line in your source code, asks "Should this expression be

`boundary<corner`*, or*

`boundary<corner+1`*?" "*

By creating a mutant of a program we are essentially challenging its developer, or the tester by asking: "Is the program correct or its mutant? Or, are the two equivalent?

Scenarios such as the one above arise during informal discussions with colleagues, and also during formal code reviews. A programmer is encouraged to show why an alternate solution to a problem is not the correct, or a better, than the one selected. The alternate solution could lead to a program significantly different from the current one. It could also be a simple mutant as in the scenario above.

There are several ways a programmer could argue in favor of the existing solution such as `boundary<corner` in the scenario above. One way is to give a performance related argument that shows why the given solution is better than the alternate proposed in terms of the performance of the application.

Some times a program and its mutant might be equivalent though one is preferred over the other for reasons such as performance or maintainability.

Another way is to show that the current solution is correct and preferred, and the proposed alternate is incorrect or not preferred. If the proposed alternate is a mutant of the original solution, then the programmer needs to simply show that the mutant behaves differently than the current solution and is incorrect. This can be done by proposing a test case and showing that the current solution and the proposed solutions are different and the current solution is correct.

It is also possible that the proposed alternate is equivalent to the current solution. However, this needs to be shown for all possible inputs and requires a much stronger argument than the one used to show that the alternate solution is incorrect.

Each mutant creates a scenario that has to be either rejected by the tester as being invalid or shown to be equivalent to the program under test.

Mutation offers a systematic way of generating a number of scenarios similar to the one described above. It places the burden of proof on the tester, or the developer that the program under test is correct while the mutant created is indeed incorrect. As we show in the remainder of this chapter, testing using mutation often leads to the discovery of subtle

flaws in the program under test. This discovery usually takes place when a tester tries to show that the mutant is an incorrect solution to the problem at hand.

8.3 Test Assessment Using Mutation

Now that we know what mutation is, and what mutants look like, let us understand how mutation is used for assessing the adequacy of a test set. The problem of test assessment using mutation can be stated as follows.

Let P be a program under test, T a test set for P, and R the set of requirements that P must meet. Suppose that P has been tested against all tests in T and found to be correct with respect to R on each test case. We want to know how good is T ?

The mutation score is a quantitative assessment of the "goodness", or adequacy, of a test set. This score is a number between 0 and 1.

Mutation offers a way of answering the question stated above. As explained later in this section, given P and T, a quantitative assessment of the goodness of T is obtained by computing a *mutation score* of T. Mutation score is a number between 0 and 1. A score of 1 means that T is adequate with respect to mutation. A score lower than 1 means that T is inadequate with respect to mutation. An inadequate test set can be enhanced by the addition of test cases that increase the mutation score.

8.3.1 A procedure for test adequacy assessment

Let us now dive into a procedure for assessing the adequacy of a test set. One such procedure is illustrated in Figure 8.1. It is important to note that Figure 8.1 shows one possible sequence of 12 steps for assessing the adequacy of a given suite of tests using mutation; other sequencings are possible and discussed later in this section.

Do not be concerned if Figure 8.1 looks confusing and rather jumbled. Also, some terms used in the figure have not been defined yet. Any confusion should be easily overcome by going through the step-by-step explanation that follows. All terms used in the figure are defined as we proceed with the explanation of the assessment process.

For the sake of simplicity, we use Program P8.1 throughout our explanation through a series of examples. Certainly, P8.1 does not represent the kind of programs for which mutation is likely to be used in commercial and research environments. However, it serves our instructional purpose in this section: that of explaining various steps in Figure 8.1. Also, and again for simplicity, we assume that Program P8.1 is correct as per its requirements. In Section 8.8, we will see how the assessment of adequacy using mutation helps in the detection of faults in the program under test.

Figure 8.1 A procedure used in the assessment of the adequacy of a test set using mutation. Solid lines point to the next process step. Dashed lines indicate data transfer between a data repository and a process step. L, D, and E denote, respectively, the set of live, distinguished, and equivalent mutants defined in the text. $P(t)$ and $M(t)$ indicate, respectively, the program and mutant behaviors observed upon their execution against test case t.

Given program P under test and test set T, mutation testing begins with the execution of P against all tests in T. If P fails on any of these tests then it ought to be corrected and run again to ensure that it behaves as expected against all tests in T.

Step 1: Program execution

The first step in assessing the adequacy of a test set T with respect to program P and requirements R, is to execute P against each test case in T. Let $P(t)$ denote the observed behavior of P when executed against t. Generally, the observed behavior is expressed as a set of values of output variables in P. However, it might also relate to the performance of P.

This step might not be necessary if P has already been executed against all elements in T and $P(t)$ recorded in a database. In any case, the end result of executing Step 1 is a database of $P(t)$ for all $t \in T$.

At this point we assume that $P(t)$ is correct with respect to R for all $t \in T$. If $P(t)$ is found to be incorrect, then P must be corrected and Step 1 executed again. The point to be noted here is that test adequacy assessment using mutation begins when P has been found to be correct with respect to the test set T.

Example 8.7 Consider Program P8.1 which we shall refer to as P. P is supposed to compute the following function $f(x, y)$.

$$f(x,y) = \begin{cases} x+y & \text{if } x<y, \\ x*y & \text{otherwise.} \end{cases}$$

Suppose we have tested P against the following set of tests.

$$T_P = \left\{ \begin{array}{ll} t_1: & <x=0, y=0>, \\ t_2: & <x=0, y=1>, \\ t_3: & <x=1, y=0>, \\ t_4: & <x=-1, y=-2> \end{array} \right\}$$

The database of $P(t)$ for all $t \in T_P$, is tabulated below.

Test case (t)	**Expected output** $f(x,y)$	**Observed output** $P(t)$
t_1	0	0
t_2	1	1
t_3	0	0
t_4	1	2

Note that $P(t)$ computes function $f(x, y)$, for all $t \in T_P$. Hence, the condition that $P(t)$ be correct on all inputs from T_P, is satisfied. We can now proceed further with adequacy assessment.

Step 2: Mutant generation

The next step in test adequacy assessment is the generation of mutants. While we have shown what a mutant looks like and how it is generated, we have not given any systematic procedure to generate a set of mutants given a program. We will do so in Section 8.4.

Once P has been found to behave correctly on all tests in T, one generates mutants of P. This step requires the selection of mutation operators. It could be executed in a big-bang manner or incrementally.

For now we assume that the following mutants have been generated from P by (a) altering the arithmetic operators such that any occurrence of the addition operator (+) is replaced by the subtraction operator (−) and that of the multiplication operator (∗) by the divide operator (/), and (b) replacing each occurrence of an integer variable v by $v + 1$.

A mutant is considered "live" until it is shown to behave differently than its parent program or proved to be equivalent to it.

Example 8.8 By mutating the program as mentioned above, we obtain a total of eight mutants, labeled M_1 through M_8, shown in the following table. Note that we have not listed the complete mutant programs for the sake of saving space. A typical mutant will look just like Program P8.1 with one of the statements replaced by the mutant in the table below.

Line	Original	Mutant ID	Mutant(s)
	`begin`		None
	`int` x, y		None
	input (x, y)		None
	`if(x<y)`	M_1	`if(x+1<y)`
		M_2	`if(x<y+1)`
	`then`		None
	output(x+y)	M_3	output(x+1+y)
		M_4	output(x+y+1)
		M_5	output(x−y)
	`else`		None
	output(x*y)	M_6	output((x+1)*y)

Line	Original	Mutant ID	Mutant(s)
		M_7	output(x*(y+1))
		M_8	output(x/y)
	`end`		None

In the table above, notice that we have not mutated the declaration, input, `then`, and `else` statements. The reason for not mutating these statements is given later in Section 8.4. Of course, markers `begin` and `end` are not mutated.

At the end of Step 2, we have a set of eight mutants. We refer to these eight mutants as *live* mutants. These mutants are live because we have not yet *distinguished* them, from the original program. We will make an attempt to do so in the next few steps. Thus we obtain the set $L = \{M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8\}$. Note that *distinguishing* a mutant from its parent is sometimes referred to as *killing* a mutant.

Steps 3 and 4: Select next mutant

After having created mutants, each mutant is selected and executed against tests in T until it shows a behavior different from that of the program under test or it has been executed against all tests.

In Steps 3 and 4 we select the next mutant to be considered. This mutant must not have been selected earlier. Notice that at this point we are starting a loop that will cycle through all mutants in L until each mutant has been selected. Obviously, we select the next mutant only if there is a mutant remaining in L. This check is made in Step 3. If there are live mutants in L that have never been selected in any previous step, then a mutant is selected arbitrarily. The selected mutant is removed from L.

Example 8.9 In Step 3 eight mutants remain in L and hence we move to Step 4 and select any one of these eight. The choice of which mutant to select is arbitrary. Hence, let us select mutant M_1. After moving M_1 from L, we have $L = \{M_2, M_3, M_4, M_5, M_6, M_7, M_8\}$

Steps 5 and 6: Select next test case

Having selected a mutant M, we now attempt to find whether or not at least one of the tests in T can distinguish it from its parent P. To do so, we need to execute M against tests in T. Thus at this point we enter another loop that is executed for each selected mutant. The loop terminates when all tests are exhausted, or M is distinguished by some test, whichever happens earlier.

Example 8.10 Now that we have selected M_1, we need to check if can be distinguished by T from P. In Step 5 we select the next test. Again, any of the tests in T_P, against which M_1 has not been executed so far, can be selected. Let us select t_1: $< x = 0, y = 0 >$.

A mutant whose behavior is observed to be different from its parent program is considered "distinguished." It is removed from further consideration.

Steps 7, 8, and 9: Mutant execution and classification

So far we have selected a mutant M for execution against test case t. In Step 7 we execute the M against t. In Step 8 we check if the output generated by executing M against t is the same or different from that generated by executing P against t.

Example 8.11 So far we have selected mutant M_1 and test case t_1. In Step 7 we execute M_1 against t_1. Given the inputs $x = 0$ and $y = 0$, condition $x + 1 < y$ is false leading to 0 as the output. Thus we see that $P(t_1) = M_1(t_1)$. This implies that test case t_1 is unable to

distinguish M_1 from P. At Step 8, the condition $P(t) = M(t)$ is true for $t = t_1$. Hence we return to Step 5.

In Steps 5 and 6 we select the next test case, t_2, and execute $M1$ against t_2. In Step 8 we notice that $P(t_2) \neq M(t_2)$ as $M(t_2) = 0$. This terminates the test execution loop that started at Step 5 and we follow Step 9. Mutant $M1$ is added to the set of distinguished (or killed) mutants.

Example 8.12 For the sake of completeness, let us go through the entire mutant execution loop that started at Step 3. We have already considered mutant M_1. Next, we select mutant M_2 and execute it against tests in T until either it is distinguished or all tests are exhausted.

The results of executing Steps 3 through 9 are summarized in the following table. Column D indicates the contents of set D of distinguished mutants. As indicated, all mutants except M_2 are distinguished by T. Initially, all mutants are live.

As per Step 8 in Figure 8.1, execution of a mutant is interrupted soon after it has been distinguished. Entries marked NE in the table below indicate that the mutant in the corresponding row was not executed against the test case in the corresponding column. Note that mutant M_8 is distinguished by t_1 because its output is undefined (indicated by the entry marked "U") due to division by 0. This implies that $P(t_1) \neq M_8(t_1)$. The first test case distinguishing a mutant is marked with an asterisk (*).

Program	t_1	t_2	t_3	t_4	D
$P(t)$	0	1	0	2	{ }
Mutant					
$M_1(t)$	0	0*	NE	NE	$\{M_1\}$
$M_2(t)$	0	1	0	2	$\{M_1\}$

Program	t_1	t_2	t_3	t_4	D
$M_3(t)$	0	2^*	NE	NE	$\{M_1, M_3\}$
$M_4(t)$	0	2^*	NE	NE	$\{M_1, M_3, M_4\}$
$M_5(t)$	0	-1^*	NE	NE	$\{M_1, M_3, M_4, M_5\}$
$M_6(t)$	0	1	0	0^*	$\{M_1, M_3, M_4, M_5, M_6\}$
$M_7(t)$	0	1	1^*	NE	$\{M_1, M_3, M_4, M_5, M_6, M_7\}$
$M_8(t)$	U^*	NE	NE	NE	$\{M_1, M_3, M_4, M_5, M_6, M_7, M_8\}$

U: Output undefined. NE: Mutant not executed. *: First test to distinguish the mutant in the corresponding row.

Step 10: Live mutants

When none of the tests in T is able to distinguish mutant M from its parent P, M is placed back into the set of live mutants. Of course, any mutant that is returned to the set of live mutants is not selected in Step 4 as it has already been selected once.

A mutant that is not distinguished by any test in T remains "live." Such a mutant could either be equivalent to its parent program or it could be distinguished by a test that is not in T.

Example 8.13 In our example, there is only one mutant, M_2, that could not be distinguished by T_P. M_2 is returned to the set of live mutants in Step 10. Notice, however, that M_2 will not be selected again for classification as it has been selected once.

Step 11: Equivalent mutants

After having executed all mutants, one checks if there are any mutants remain live, i.e. set L is non-empty. Any remaining live mutants are tested for equivalence to their parent program. A mutant M is considered equivalent to its parent P if for each test input from the input domain of P, the observed behavior of M is identical to that of P. We discuss the issue of determining whether or not a mutant is equivalent, in Section 8.7. The following example illustrates the process with respect to P8.1.

A mutant is considered equivalent to its parent P if there exists no test in the input domain of P that causes P and M to generate different outputs.

To obtain an accurate mutation score for test T, one needs to determine which of the live mutants are equivalent to the parent program.

Example 8.14 Note from Example 8.13 that no test in T_P is able to distinguish M_2 from P and hence M_2 is live. We ask: is M_2 *equivalent to* P? An answer to such questions requires careful analysis of the behavior of the two programs: the parent and its mutant. In this case, let $f_P(x,y)$ be the function computed by P and $g_{M_2}(x,y)$ that computed by M_2. The two functions follow.

$$f_P(x,y) = \begin{cases} x+y & \text{if } x<y, \\ x*y & \text{otherwise.} \end{cases}$$

$$g_{M_2} = \begin{cases} x+y & \text{if } x<y+1, \\ x*y & \text{otherwise.} \end{cases}$$

Rather than show that M_2 is equivalent to P, let us first attempt to find $x = x_1$ and $y = y_1$ such that $f_P(x_1,y_1) \neq g_{M_2}(x_1,y_1)$. A simple examination of the two functions reveals that the following two conditions, denoted as C_1 and C_2, must hold for $f_P(x_1,y_1) \neq g_{M_2}(x_1,y_1)$.

$$C_1 : (x_1 < y_1) \neq (x_1 < y_1 + 1)$$

$$C_2 : x_1 * y_1 \neq x_1 + y_1$$

For the conjunct of C_1 and C_2 to hold, we must have $x_1 = y_1 \neq 0$. We note that while test case t_1 satisfies C_1, it fails to satisfy C_2. However, the following test case does satisfy both C_1 and C_2.

$t :< x = 1, y = 1 >$

A simple computation reveals that $P(t) = 1$ and $M_2(t) = 2$. Thus we have shown that M_2 can be distinguished by at least one test case.

Hence, M_2 is not equivalent to its parent P. As there is only one live mutant in L that we have already examined for equivalence, we are done with Step 11. Set E of equivalent mutants remains empty.

Step 12: Computation of mutation score

This is the final step in the assessment of test adequacy. Given sets L, D, and E, the mutation score, denoted by $MS(T)$ of a test set T is computed as follows.

$$MS(T) = \frac{|D|}{|L| + |D|}.$$

The mutation score of a test set T is the ratio of the number of distinguished mutants to the total number of non-equivalent mutants. Thus, the mutation score is between 0 and 1 (inclusive).

It is important to note that set L contains only the live mutants, none of which is equivalent to the parent. As is evident from the formula above, a mutation score is always between 0 and 1.

Given that $|M|$ denotes the total number of mutants generated in Step 2, the following definition of mutation score is also used.

$$MS(T) = \frac{|D|}{|M| - |E|}.$$

If a test set T distinguishes all mutants, except those that are equivalent, then $|L| = 0$ and the mutation score is 1. If T does not distinguish any mutant, then $|D| = 0$ and the mutation score is 0.

An interesting case arises when a test set does not distinguish any mutant and all mutants generated are equivalent to the original program. In this case, $|D| = |L| = 0$ and the mutant score is undefined. Hence we ask, *Is the test set hopelessly inadequate*? The answer: *No it is not.* In this case, the set of mutants generated is insufficient to assess the adequacy of the test set. In practice, it is rare, if not impossible, to find a situation where $|L| = |D| = 0$ and all the generated mutants are equivalent to the parent. This fact will be evident when we discuss mutation operators in Section 8.4.

Though rare, it is possible to generate mutants all of which are equivalent to the parent program.

There is no "golden" mutation score for a test set. The score will depend on the goodness of the test set and the nature of mutants generated. One could generate few mutants and obtain a high mutation score. This leads to the importance of selecting the mutation operators.

Note that a mutation score for a test set is computed with respect to a given set of mutants. Thus, there is no "golden" mutation score that remains fixed for a given test set. One will likely obtain different mutation scores for a test set evaluated against a different set of mutants. Thus, while a test set T might be perfectly adequate, i.e. $MS(T) = 1$ with respect to a given set of mutants, it might be perfectly inadequate, i.e. $MS(T) = 0$ with respect to another.

Example 8.15 Continuing with the illustrative example, it is easy to compute the mutation score for T_P as given in Example 8.7. At the start, there are eight mutants in Example 8.8. At the end of Step 11 there is one live mutant, seven distinguished mutants, and no equivalent mutant. Thus, $|D| = 7$, $|L| = 1$, and $|E| = 0$. This gives us $MS(T_P) = 7/(7+1) = 0.875$.

Test case t in Example 8.14 distinguishes the lone live mutant M_2 from its parent. Suppose t is added to T_P such that the test set T'_P so modified now contains five test cases. What is the revised mutation score for T'_P? Obviously, $MS(T'_P) = 1$. However, T_P has been enhanced by adding t.

8.3.2 Alternate procedures for test adequacy assessment

An incremental approach to mutation testing might be a better option than the big-bang approach where all mutants are generated and executed near-simultaneously.

Several variations of the procedure described in the previous section are possible, and often recommended. Let us start at Step 2. As indicated in Figure 8.1, Step 2 implies that all mutants are generated before test execution. However, an incremental approach might be better suited given the large number of mutants that will likely be created even for short programs, or for short functions within an application.

Using an incremental approach for the assessment of test adequacy, one generates mutants from only a subset of the mutation operators. The given test set is evaluated against this subset of mutants and a mutation score computed. If the score is less than 1, additional tests are created to ensure a score close to 1. This cycle is repeated with additional subsets of mutant operators and the test set further enhanced. The process may continue until all mutant operators have been considered.

The incremental approach allows an incremental enhancement of the test set at hand. This is in contrast to enhancing the test set at the end of the assessment process when a large number of mutants might remain live thereby overwhelming a tester.

It is also possible to use a multi-tester version of the process illustrated in Figure 8.1. Each tester can be assigned a subset of the mutation operators. The testers share a common test set. Each tester computes the adequacy with respect to the mutants generated by the operators assigned

and enhances the test set. New tests so created are made available to the other testers through a common test repository. While this method of concurrent assessment of test adequacy might reduce the time to develop an adequate test set, it might lead to redundant tests (see Exercise 8.11).

8.3.3 "Distinguished" versus "killed" mutants

As mentioned earlier, a mutant that behaves differently than its parent program on some test input is considered *killed*. However, for the sake of promoting a sense of calm and peace during mutation testing, we prefer to use the term *distinguished* in lieu of *killed*. A cautionary note: most literature on mutation testing uses the term "killed." Some people prefer to say that "a mutant has been detected" implying that the mutant referred to is distinguished from its parent.

Most literature in mutation testing refers to a "distinguished" mutant as a "killed" mutant and the act of "distinguishing a mutant" as "killing a mutant." Which term to use is a personal preference.

Of course, a "distinguished mutant" is not *distinguished* in the sense of a "distinguished professor" or a "distinguished lady." It is considered "distinguished from its parent by a test case." Certainly, if a program mutant could feel and act like humans do, it probably will be happy to be considered "distinguished" like some of us do. Other possible terms for a distinguished mutant include *extinguished* and *terminated*.

8.3.4 Conditions for distinguishing a mutant

A test case t that distinguishes a mutant M from its parent P must satisfy the following three conditions labeled as C_1, C_2, and C_3.

1. C_1: *Reachability*: Must force the execution of a path from the start statement of M to the point of the mutated statement.
2. C_2: *State infection*: Must cause the state of M and P to differ consequent to some execution of the mutated statement.
3. C_3: *State propagation*: Must ensure that the difference in the states of M and P, created due to the execution of the mutated statement, propagates until after the end of mutant execution.

For a mutant to be distinguished under strong mutation, a test case must force the control to reach the mutated statement, cause the state of the mutant to become "infected," i.e. be different from that of its parent, and the infected state must propagate to the output when the mutant terminates such that the outputs of the mutant and its parent are observably different.

Thus a mutant is considered distinguished only when test t satisfies $C_1 \wedge C_2 \wedge C_3$. To be more precise, condition C_1 is necessary for C_2 which in turn is necessary for C_3. The following example illustrates the three conditions. Note that condition C_2 need not be true during the first execution of the mutated statement, though it must hold during some execution. Also, all program variables, and the location of program control, constitute program state at any instant during program execution.

A mutant is considered *equivalent* to the program under test if there is no test case in the input domain of P that satisfies each of the three conditions above. It is important to note that equivalence is based on identical

behavior of the mutant and the program over the entire input domain and not just the test set whose adequacy is being assessed.

Example 8.16 Consider mutant M1 of Program P8.2. The reachability condition requires that control arrive at line 14. As this is a straight line program, with a `return` at the end, every test case will satisfy the reachability condition.

The state infection condition implies that after execution of the statement at line 14, the state of the mutant must differ from that of its parent. Let $\texttt{danger}_P$ and $\texttt{danger}_M$ denote the values of variable *danger* soon after the execution of line 14. The state infection condition is satisfied when $\texttt{danger}_P \neq \texttt{danger}_M$. Any test case for which $\texttt{highcount} = 3$ satisfies the state infection condition.

Finally, as line 14 is the last statement in the mutant that can effect the state, no more changes can occur to the value of `danger`. Hence the state propagation condition is satisfied trivially by any test case that satisfies the state infection condition. Following is a test case that satisfies all three conditions.

$$t : < \texttt{currentTemp} = [20, 34, 29], \texttt{maxTemp} = 18 >$$

For test case t, we get $P(t) = \texttt{veryHigh}$ and $M(t) = \texttt{none}$ thereby distinguishing the mutant from its parent.

Example 8.17 Next consider function `findElement`, in P8.3, that searches for a chemical element whose atomic weight is greater than `w`. The function is required to return the name of the first such element found amongst the first `size` elements, or return "None" if no such element is found. Arrays `name` and `atWeight` contain, respectively, the element names and the corresponding atomic weights. A global constant `maxSize` is the size of the two arrays.

Program P8.3

```
1   String findElement (name, atWeight, int size,
    float~w){
2     String name[maxSize]; float atWeight[maxSize], w;
3     int index=0;
4     while (index<size){
5      if (atWeight[index]>w)
6       return(name[index]);
7      index=index+1;
8     }
9     return("None");
10  }
```

Consider a mutant of P8.3 obtained by replacing the loop condition by `index≤size`.

The reachability condition to distinguish this mutant is satisfied by any test case that causes `findElement` to be invoked. Let C_P and C_M denote, respectively, the loop conditions in P8.3 and its mutant. The state infection condition is satisfied by a test case for which the relation $C_P \neq C_M$ holds during some loop iteration.

The state propagation condition is satisfied when the value returned by the mutant differs from the one returned by its parent. The following test satisfies all three conditions.

t_1 : < `name` = ["Hydrogen", "Nitrogen", "Oxygen"],
`atWeight` = [1.0079, 14.0067, 15.9994],
`maxSize` = 3, `size` = 2, `w` = 15.0 >

Upon execution against t_1, the loop in P8.3 terminates unsuccessfully and the program returns the string "None." On the contrary, when executing the mutant against t_1, the loop continues until `index` = `size` when it finds an element with atomic weight greater than `w`. While P8.3 returns the string "None," the mutant returns "Oxygen." Thus t_1 satisfies all three conditions for distinguishing the mutant from its parent. (Also see Exercises 8.8 and 8.10.)

8.4 Mutation Operators

In the previous sections, we gave examples of mutations obtained by making different kinds of changes to the program under test. Up until now, we have generated mutants in an ad hoc manner. However, there exists a systematic method, and a set of guidelines, for the generation of mutants that can be automated. One such systematic method and a set of guidelines are described in this section. We begin our discussion by learning what is a mutation operator and how is it used.

A mutation operator is applied to a program to generate zero or more mutants.

A mutation operator is a generative device. Other names used for mutation operators include *mutagenic operator*, *mutant operator*, and simply

Figure 8.2 A set of k mutants $M_1, M_2, \ldots, M_k$ of P generated by applying a mutation operator. The number of mutants generated, k, depends on P and the mutation operator; k could be 0.

operator. In the remainder of this chapter, the terms "mutation operator" and "operator" are used interchangeably when there is no confusion.

Each mutation operator is assigned a unique name for ease of reference. For example, ABS and ROR are two names of mutation operators whose functions are explained in Example 8.18.

A mutation operator must generate a syntactically correct program; one that compiles without errors.

As shown in Figure 8.2, when applied to a syntactically correct program *P*, a mutation operator generates a set of syntactically correct mutants of *P*. We apply one or more mutation operators to *P* to generate a variety of mutants. A mutation operator might generate no mutants or one or more mutants. This is illustrated in the next example.

Example 8.18 Consider an operator named CCR. When applied to a program, CCR generates mutants by replacing each occurrence of a constant *c* by some other constant *d*; both *c* and *d* must appear in *P*. When applied to P8.1, CCR will not generate any mutant as the program does not contain any constant.

Next, consider an operator named ABS. When applied, ABS generates mutants by replacing each occurrence of an arithmetic expression *e* by the expression abs(*e*). Here it is assumed that *abs* denotes the "absolute value" function found in many programming languages. The following eight mutants are generated when ABS is applied to P8.1.

Location	Statement	Mutant
Line 4	`if`(x<y)	`if`(abs(x)<y)
		`if`(x<abs(y))
Line 5	output(x+y)	output(abs(x)+y)
		output(x+abs(y))
		output(abs(x+y))
Line 7	output(x*y)	output(abs(x)*y)
		output(x*abs(y))
		output(abs(x*y))

Note that the input statement and declarations have not been mutated. Declarations are not mutated and are considered the source for variables names and types to be used for certain types of mutations discussed later in this section. The input statement has not been mutated as adding an *abs* operator to an input value will lead to a syntactically incorrect program.

8.4.1 Operator types

Mutation operators are designed to model simple programming mistakes that programmers make. Faults in programs could be much more complex than the simple mistakes modeled by a mutation operator. However, it has been found that, despite the simplicity of mutations, complex faults are discovered while trying to distinguish mutants from their parent. We return to this suspicious sounding statement in Section 8.6.2.

Mutation operators model simple programming mistakes. Thus, these operators are language dependent.

It is to be noted that some mutation tools provide mutation operators designed explicitly to enforce code and domain coverage. The STRP is one such operator for languages C and Fortran. When applied to a program, it creates one mutant for each statement by replacing the statement with a *trap* condition. Such a mutant is considered distinguished from its parent when control arrives at the replaced statement. A test set that distinguishes all STRP mutants is considered adequate with respect to the statement coverage criterion.

The VDTR operator for C ensures that the given test set is adequate with respect to domain coverage for appropriate variables in the program under test. The domain is partitioned into a set of negative values, 0, and positive values. Thus, for an integer variable, domain coverage is obtained, and the corresponding mutants distinguished, by ensuring that the variables takes on any negative value, 0, and any positive value, during some execution of the program under test.

While a set of mutation operators is specific to a programming language, it can nevertheless be partitioned into a small set of generic categories. One such categorization appears in Table 8.1. While reading the examples in the table, read the right arrow (→) as *mutates to*. Also, note that in the first entry, constant 1 has been changed to 3, as constant 3 appears in another statement (`y=y*3`).

Mutation operators can be categorized with respect to, for example, the generic syntactic element in the program on which they are applicable.

Only a set of four generic categories are shown in Table 8.1. As shown later in this section, in practice, there are many mutation operators under each category. The number and type of mutation operators under each category depend on the programming language for which the operators are designed.

The constant mutations category consists of operators that model mistakes made in the use of constants. Constants of different types, such as floats, integers, and booleans, are used in mutations. The domain of any mutation operator in this category is the set of constants appearing in the program that is mutated; a mutation operator does not invent constants. Thus, in the example that appears in Table 8.1, statement `x=x+1` has been mutated to `x=x+3` because constant 3 appears in another statement, i.e. in `y=y*3`.

The operator mutations category contains mutation operators that model common mistakes made in relation to the use of operators. Mistakes modeled include the incorrect use of arithmetic operators, relational operators, logical operators, and any other types of operators the language provides.

Some mutation operators generate only a few mutants while others a lot. For example, the operator that replaces a program variable by another program variable is likely to generate a large number of mutants when compared with the operator that replaces a program constant by another.

The statement mutation category consists of operators that model various mistakes made by programmers in the placement of statements in a program. These mistakes include incorrect placement of a statement, missing statement, incorrect loop termination, and incorrect loop formation.

The variable mutations category contains operators that model a common mistake programmers make when they incorrectly use a variable while formulating a logical or an arithmetic expression. Two incorrect uses are illustrated in Table 8.1. The first is the use of an incorrect variable `x`, a `y` should have been used instead. The second is an incorrect use of `x`, its absolute value should have been used. As with constant mutation operators, the domain of variable mutation operators includes all variables declared and used in the program being mutated. Mutation operators do not manufacture variables!

8.4.2 Language dependence of mutation operators

While it is possible to categorize mutation operators into a few generic categories, as in the previous section, the operators themselves are dependent

Table 8.1 A sample of generic categories of mutation operators and the common programming mistakes modeled.

Category	Mistake modeled	Examples
Constant mutation	Incorrect constant	`x=x+1;` → `x=x+3;` `y=y*3;` → `y=y*1;`
Operator mutations	Incorrect operator	`if(x<y)` → `if(x≤y)` `x++` → `++x`
Statement mutation	Incorrectly placed statement	`z=x+1;` → delete `z=x+1;` `break;` → `break` `z=x+1;`
Variable mutations	Incorrectly used variable	`z=x+1;` → `z=y+1;` `z=x+y;` → `z=abs(x)+y;`

on the syntax of the programming language. For example, for a program written in ANSI C (hereafter referred to as C), one needs to use mutation operators for C. A Java program is mutated using mutation operators designed for the Java language.

There are at least three reasons for the dependence of mutation operators on language syntax. First, given that the program being mutated is syntactically correct, a mutation operator must produce a mutant that is also syntactically correct. To do so requires that a valid syntactic construct be mapped to another valid syntactic construct in the same language.

Mutation operators exist for a variety of languages including C, C++, Java, Lisp, Python, and Fortran.

Second, the domain of a mutation operator is determined by the syntax rules of a programming language. For example, in Java, the domain of a mutation operator that replaces one relational operator by another is $\{<, <=, >, >=, !=, ==\}$.

Third, peculiarities of language syntax have an effect on the kind of mistakes that a programmer could make. Note that aspects of a language such as procedural versus object oriented, are captured in the language syntax.

Example 8.19 The Access Modifier Change (AMC) operator in Java replaces one access modifier, e.g. `private`, by another, e.g. `public`. This mutation models mistakes made by Java programmers in controlling the access of variables and methods.

While incorrect scoping is possible in C, the notion of access control is made explicit in Java though a variety of access modifiers and their combinations. Hence the AMC operator is justified for mutating Java programs. There is no matching operator for mutating C programs due to the absence of explicit access control; access control in C is implied by the location of declarations and can incorrect the placement of declarations often be checked by a compiler.

Example 8.20 Suppose `x`, `y`, and `z` are integer variables declared appropriately in a C program. The following is a valid C statement:

S: `z=(x<y) ? 0: 1;`

Suppose now that we wish to mutate the relational operator in *S*. The relational operator $<$ can be replaced by any one of the remaining five relational operators in C, namely $==$, $!=$, $>$, $<=$, and $>=$. Such a replacement would produce a valid statement in C, and in many other languages too given that the statement to be mutated is valid in that language.

However, C also allows the presence of any arithmetic operator in place of the $<$ operator in the statement above. Hence, the $<$

can also be replaced by any addition and multiplication operator from the set $\{+, -, *, /, \%\}$. Thus, for example, while the replacement of $<$ in S by the operator $+$ will lead to a valid C statement, this would not be allowed in several other languages such as in Java and Fortran. This example illustrates that the domain of a mutation operator that mutates a relational operator is different in C than it is in Java.

This example also illustrates that the types of mistakes made by a programmer using C are different from those that a programmer might make, for instance, when using Java. It is perfectly legitimate to think that a C programmer has S as

S': `z=(x+y) ? 0: 1;`

for whatever reasons, whereas the intention was to code it as S. While both S and S' are syntactically correct, they might to different program behavior, only one of them being correct. It is easy to construct a large number of such examples of mutations, and mistakes, that a programmer might be able to make in one programming language but not in another.

The example above suggests that the design of mutation operators is a language dependent activity. Further, while it is possible to design mutation operators for a given programming language through guesses on what simple mistakes might a programmer make, a scientific approach is to base the design on empirical data. Such empirical data consolidates the common programming mistakes. In fact mutation operators for a variety of programming languages have been designed based on a mix of empirical evidence and collective programming experience of the design team.

8.5 Design of Mutation Operators

8.5.1 Goodness criteria for mutation operators

Mutation is a tool to assess the goodness of a test set for a given program. It does so with the help of a set of mutation operators that mutate the program under test into an often large number of mutants. Suppose that a test set T_P for program P is adequate with respect to a set of mutation operators M. What does this adequacy imply regarding the correctness of P? This question leads us to the following definition.

Ideal set of mutation operators: Let $\mathcal{P}_L$ denote the set of all programs in some language L. Let M_L be a set of mutation operators for L. M_L is

considered ideal if (a) for any $P \in \mathcal{P}$, and any test set T_P against which P is found to be correct with respect to its specification S, the adequacy of T_P with respect to M_L implies correctness of P with respect to S and (b) there does not exist any M'_L smaller in size than M_L for which property (a) is satisfied.

An ideal set of mutation operators is one that when applied to a program P under test will generate mutants such that a test set that distinguishes all these mutants without causing P to fail implies that P is correct. In practice, such a set of operators is impossible to derive.

Thus, an ideal set of mutant operators is minimal in size and ensures that adequacy with respect to this set implies correctness. While it is impossible to construct an ideal set of mutation operators for any but the most trivial of languages, it is possible to construct one that ensures correctness with relatively high probability. It is precisely such set of mutation operators that we desire. The implication of probabilistic correctness is best understood with respect to the "competent programmer hypothesis" and the "coupling effect" explained in Sections 8.6.1 and 8.6.2.

Properties (a) and (b), given in the definition of an ideal set of mutation operators, relate to the fault detection effectiveness and the cost of mutation testing. We seek a set of mutation operators that would force a tester into designing test cases that reveal as many faults as possible. Thus we desire high fault detection effectiveness. However, in addition, we would like to obtain high fault detection effectiveness at the lowest cost measured in terms of the number of mutants to be considered. The tasks of compiling, executing, and analyzing mutants are time consuming and must be minimized. The cost of mutation testing is generally lowered by a reduction in the number of mutants generated.

8.5.2 Guidelines

The guidelines here serve two purposes. One, they provide a basis on which to decide whether or not something should be mutated. Second, they help in understanding and critiquing an existing set of mutation operators such as the ones described later in this chapter.

It is to be noted that the design of mutation operators is as much of an art as it is science. Extensive experimentation, and experience with other mutation systems, is necessary to determine the goodness of a set of mutation operators for some language. However, to gain such experience, one needs to develop a set of mutation operators.

Empirical data on common mistakes serves as a basis for the design of mutation operators. In the early days of mutation research, mutation operators were designed based on empirical data available from various error studies. The effectiveness of these operators has been studied extensively. The guidelines provided here are based on past error studies, experience with mutation systems, and empirical studies to assess how effective are mutation operators in detecting complex errors in programs.

1. *Syntactic correctness*: A mutation operator must generate a syntactically correct program.
2. *Representativeness*: A mutation operator must model a common fault that is simple. Note that real faults in programs are often not simple. For example, a programmer might need to change significant portions of the code in order to correct a mistake that led to a fault. However, mutation operators are designed to model only simple faults, many of them taken together make up a complex fault.
   We emphasize that simple faults, e.g. a < operator mistyped as a > operator, could creep in due to a variety of reasons. However, there is no way to guarantee that for all applications, a simple fault will not lead to a disastrous consequence, e.g. a multimillion dollar rocket failure. Thus the term "simple fault" must not be equated with "simple," "inconsequential," or "low severity" failures.
3. *Minimality and effectiveness*: The set of mutation operators should be as small and effective as possible.
4. *Precise definition*: The domain and range of a mutation operator must be clearly specified; both depend on the programming language at hand. For example, a mutation operator to mutate a binary arithmetic operator, e.g. +, will have + in its domain as well in the range. However, in some languages, e.g. in C, replacement of a binary arithmetic operator by a logical operator, e.g. && will lead to a valid program. Thus all such syntax preserving replacements must be considered while defining the range.

One ought to be careful when designing mutation operators for a specific programming language. The guidelines here are intended to assist in the design of mutation operators.

8.6 Founding Principles of Mutation Testing

Appropriate use of mutation testing can lead to close-to-correct programs.

Mutation testing is a powerful testing technique for achieving correct, or close to correct, programs. It rests on two fundamental principles. One principle is commonly known as the *competent programmer hypothesis* (CPH), or *the competent programmer assumption*. The other is known as the *coupling effect*. We discuss these next.

8.6.1 The competent programmer hypothesis

The CPH arises from a simple observation made of practicing programmers. The hypothesis states that given a problem statement, a programmer writes a program P that is in the general neighborhood of the set of correct programs.

An extreme interpretation of CPH is that when asked to write a program to find the account balance, given an account number, a programmer is unlikely to write a program that deposits money into an account.

Of course, while such a situation is unlikely to arise, a devious programmer might certainly write such a program.

A more reasonable interpretation of the CPH is that the program written to satisfy a set of requirements will be a few mutants away from a correct program. Thus, while the first version of the program might be incorrect, it could be corrected by a series of simple mutations. One might argue against the CPH by claiming something like "What about a missing condition as the fault? One would need to add the missing condition in order to arrive at a correct program." Indeed, given a correct program P_c, one of its mutants is obtained by removing the condition from a conditional statement. Thus a missing condition does correspond to a simple mutant.

The CPH assumes that the programmer knows of an algorithm to solve the problem at hand, and if not, will find one prior to writing the program. It is thus safe to assume that when asked to write a program to sort a list of numbers, a competent program knows of, and makes use of, at least one sorting algorithm. Certainly, mistakes could be made while coding the algorithm. Such mistakes will lead to a program that can be corrected by applying one or more first order mutations.

Competent Programmer Hypothesis is the foundation of mutation testing. According to this hypothesis competent programmers write programs that are not too far away from correct programs in terms of respective syntax.

8.6.2 The coupling effect

While the CPH arises out of observations of programmer behavior, the coupling effect is observed empirically. The coupling effect has been paraphrased by DeMillo, Lipton, and Sayward as follows.

Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors.

Stated alternately, again in the words of DeMillo, Lipton, and Sayward "...seemingly simple tests can be quite sensitive via the coupling effect." As explained earlier, a "seemingly simple" first order mutant could be either equivalent to its parent or not. For some input, a non-equivalent mutant forces a slight perturbation in the state space of the program under test. This perturbation takes place at the point of mutation and has the potential of infecting the entire state of the program. It is during an analysis of the behavior of the mutant in relation to that of its parent that one discovers complex faults.

Extensive experimentation has revealed that a test set adequate with respect to a set of first order mutants, and is very close to being adequate with respect to second order mutants. Note that it may be easy to discover a fault that is a combination of many first order mutations. Almost any test will likely discover such a fault. It is the subtle faults that are close to first order mutations that are often difficult to detect. However, due to the

The coupling effect indicates that tests that distinguish all mutants are very likely to reveal complex errors. This also implies that first order mutation is sufficient to obtain a high level of program correctness.

coupling effect, a test set that distinguishes first order mutants is likely to cause an erroneous program under test to fail. This error detection aspect of mutation testing is explained in more detail in Section 8.8.

8.7 Equivalent Mutants

Given a mutant M of program P, we say that M is equivalent to P if $P(t) = M(t)$ for all possible test inputs t. In other words, if M and P behave identically on all possible inputs, then the two are equivalent.

The meaning of "behave identically" should be considered carefully. In strong mutation, the behavior of a mutant is compared with that of its parent at the end of their respective executions. Thus, for example, an equivalent mutant might follow a path different from that of its parent but the two might produce an identical output at termination.

A mutant that is equivalent under strong mutation might be distinguished from its parent under weak mutation. This is because in weak mutation the behavior of a mutant and its parent is generally compared at some intermediate point of execution.

The problem os deciding whether a mutant is equivalent to its parent is in general unsolvable. This problem is similar to that of determining whether a path in a program is feasible or not.

The general problem of determining whether or not a mutant is equivalent to its parent is undecidable and equivalent to the halting problem. Hence, in most practical situations, determination of equivalent mutants is done by the tester through careful analysis. Some methods for the automated detection of equivalent mutants are pointed under bibliographic notes.

It should be noted that the problem of deciding the equivalence of a mutant in mutation testing is analogous to that of deciding whether or a given path is infeasible in, say, MC/DC or data flow testing. Hence, it is unwise to consider the problem of isolating equivalent mutants from non-equivalent ones as something that makes mutation testing less attractive than any other form of coverage-based assessment of test adequacy.

8.8 Fault Detection Using Mutation

Despite the simplicity of the first order mutants, mutation testing can reveal nearly all types of faults in a program.

Mutation offers a quantitative criterion to assess the adequacy of a test set. A test set T_P for program P, inadequate with respect to a set of mutants, offers an opportunity to construct new tests that will hopefully exercise P in ways different from those it has already been exercised. This in turn raises the possibility of detecting hidden faults that so far have remained undetected despite the execution of P against T_P. In this section, we show how mutants force the construction of new tests that reveal faults not necessarily modeled directly by the mutation operators.

We begin with an illustrative example that shows how a missing condition fault is detected in an attempt to distinguish a mutant created using a mutation operator analogous to the VLSR operator in C.

Example 8.21 Consider a function named `misCond`. It takes zero or more sequence of integers in array `data` as input. It is required to sum all integers in the sequence starting from the first integer and terminating at the first 0. Thus, for example, if the input sequence is (6 5 0), then the program must output 11. For the sequence (6 5 0 4), `misCond` must also output 11.

```
int misCond(data, link, F){
int data[3], link [3], F;          ← Inputs.
int sum;          ← Output.
int L;          ← Local variable.
sum=0;
L=F;
 if(L≠ −1){          ← Part of the condition is missing.
  while (L≠ −1){
   if(data[L]≠ 0) sum=sum+data[L];      ← Redundant
                                          condition.
   L=link[L];
  };
}
return(sum);
}
```

Array `link` specifies the starting location of each sequence in `data`. Array subscripts are assumed to begin at 0. An integer `F` points to the first element of a sequence in `data` to be summed. `link`(`F`) points to the second element of this sequence, `link`(`link`(`F`)) to the third element, and so on. A −1 in a `link` entry implies the end of a sequence that begins at `data`(`F`).

Sample input data is shown below. `data` contains the two sequences (6 5 0) and (6 5 0 4). `F` is 0 indicating that the function needs to sum the sequence starting at `data`(`F`) which is (6 5 0). Setting `F` to 3 specifies the second of the two sequences in `data`.

Array index	→	0	1	2	3	4	5	6
`data`	=	6	5	0	6	5	0	4
`link`	=	1	2	−1	4	5	6	−1
`F`	=	0						

In the sample inputs shown above, the sequence is stored in contiguous locations in `data`. However, this is not necessary and `link` can be used to specify an arbitrary storage pattern.

Function `misCond` has a missing condition error. The condition at line 7 should be

$((\texttt{L} \neq -1)$ `and` $(\texttt{data[L]} \neq 0))$.

Let us consider a mutant M of `misCond` created by mutating line 9 to the following:

```
if(data[F] ≠ 0) sum = sum + data[L];
```

A mutant M is considered "error-revealing" when a test that distinguishes M also reveals an error in its parent program.

We will show that M is an error-revealing mutant. Notice that variable `L` has been replaced by `F`. This is a typical mutant generated by several mutation testing tools when mutating with the variable replacement operator, e.g. the VLSR operator in C.

We will now determine a test case that distinguishes M from `misCond`. Let C_P and C_M denote the two conditions $\texttt{data(L)} \neq 0$ and $\texttt{data(F)} \neq 0$, respectively. Let $\texttt{SUM}_P$ and $\texttt{SUM}_M$ denote, respectively, the values of `SUM` when control reaches the end of `misCond` and M. Any test case t that distinguishes M must satisfy the following conditions:

1. *Reachability*: There must be a path from the start of `misCond` to the mutated statement at line 9.
2. *State infection*: $C_P \neq C_M$ must hold at least once through the loop.
3. *State propagation*: $\texttt{SUM}_P \neq \texttt{SUM}_M$.

We must have $\texttt{L} = \texttt{F} \neq -1$ for the reachability condition to hold. During the first iteration of the loop we have `F=L` due to the initialization statement immediately preceding the start of the loop. This initialization forces $C_P = C_M$ during the first iteration. Therefore, the loop must be traversed at least twice implying that $\texttt{link(F)} \neq -1$. Any one of the following two conditions could be true during the second or subsequent loop traversals.

$$\texttt{if data(F)} \neq 0 \quad \text{then} \quad \texttt{data(L)} = 0 \tag{8.1}$$

$$\texttt{if data(F)} = 0 \quad \text{then} \quad \texttt{data(L)} \neq 0 \tag{8.2}$$

However, (8.1) will not guarantee state propagation because adding 0 to `SUM` will not alter its value from the previous iteration. Condition (8.2) will guarantee state propagation but only if the sum

of the second and any subsequent elements of the sequence being considered is non-zero.

In summary, a test case t must satisfy the following conditions for `misCond(t)` $\neq M(t)$:

$$\begin{aligned} \texttt{F} &\neq -1 \\ \texttt{link(F)} &\neq -1 \\ \texttt{data(F)} &= 0 \\ \sum_j \texttt{data}(j) &\neq 0 \text{ where } j \text{ varies from } \texttt{link(F)} \text{ to the index of the last element in the sequence.} \end{aligned}$$

Suppose P_c denotes the correct version of `misCond`. It is easy to check that for any t that satisfies the four conditions above, we get $P_c(t) = 0$, whereas the incorrect function `misCond(t)` $\neq 0$. Hence test case t causes `misCond` to fail thereby revealing the existence of a fault. A sample error-revealing test case follows.

Array index	→	0	1	2
`data`	=	0	5	0
`link`	=	1	2	−1
`F`	=	0		

Exercises 8.19 and 8.20 provide additional examples of error revealing mutants. Exercise 8.21 is designed to illustrate the strength of mutation with respect to path oriented adequacy criteria.

In the above example, we have shown that an attempt to distinguish the variable replacement operator forces the construction of a test case that causes the program under test to fail. We now ask: Are there other mutations of `misCond` that might reveal the fault? In general, such questions are difficult, if not impossible, to answer without the aid of a tool that automates the mutant generation process. Next we formalize the notion of an error revealing mutant such as the one we have seen in the previous example.

8.9 Types of Mutants

We now provide a formalization of the error detection process exemplified above. Let P denote the program under test that must conform to specification S. $\mathcal{D}$ denotes the input domain of P derived from S. Each mutant

of P has the potential of revealing one or more possible errors in P. However, for one reason or another it may not reveal any error. From a tester's point of view, we classify a mutant into one of three types: error revealing, error hinting, and reliability indicating. Let P_c denote a correct version of P. Consider the following three types of mutants.

A mutant M is said to be of type *error revealing* ($\mathcal{E}$) for program P if and only if $\forall t \in \mathcal{D}$ such that $P(t) \neq M(t), P(t) \neq P_c(t)$ and that there exists at least one such test case. t is considered to be an error revealing test case.

A mutant that is equivalent to its parent program but to the correct program is considered an "error-hinting" mutant.

A mutant M is said to be of type *error hinting* ($\mathcal{H}$), if and only if $P \equiv M$ and $P_c \not\equiv M$.

A mutant M is said to be of type *reliability indicating* ($\mathcal{R}$) if and only if $P(t) \neq M(t)$ for some $t \in \mathcal{D}$ and $P_c(t) = P(t)$.

A mutant that is distinguished from its parent by test t but does not cause the parent program to fail on t, is considered "reliability indicating."

Let S_x denote the set of all mutants of type x. From the definition of equivalence, we have $S_{\mathcal{E}} \cap S_{\mathcal{H}} = \emptyset$ and $S_{\mathcal{H}} \cap S_{\mathcal{R}} = \emptyset$. A test case that distinguishes a mutant either reveals the error in which case it belongs to $S_{\mathcal{E}}$, or else it does not reveal the error in which case it belongs to $S_{\mathcal{R}}$. Thus $S_{\mathcal{E}} \cap S_{\mathcal{R}} = \emptyset$.

During testing a tool such as MuJava or Proteum generates mutants of P and executes them against all tests in T. It is during this process that one determines the category to which a mutant belongs. There is no easy, or automated way, to find which of the generated mutants belongs to which of the three classes mentioned above. However, experiments have revealed that if there is an error in P, then with high probability at least one of the mutants is error revealing.

8.10 Mutation Operators for C

In this section, we take a detailed look at the mutation operators designed for the C programming language. As mentioned earlier, the entire set of 77 mutation operators is divided into four categories: constant mutations, operator mutations, statement mutations, and variable mutations. The contents of this section should be particularly useful to those undertaking the task of designing mutation operators and developing tools for mutation testing.

A comprehensive set of mutation operators exists for the C programming language. This set was implemented in a tool named Proteum.

The set of mutation operators introduced in this section was designed at Purdue University by a group of researchers led by Richard A. Demillo. This set is perhaps the largest, most comprehensive, and the only set of of mutation operators known for C. Josè Maldonado's research group at the University of Saõ Carlos at Saõ Carlos, Brazil, has implemented the complete set of C mutation operators in a tool named Proteum. In Section 8.12,

we compare the set of mutation operators with those for some other languages. Section 8.15 points to some tools to assist a tester in mutation testing.

8.10.1 What is not mutated?

Every mutation operator has a possibly infinite domain on which it operates. The domain itself consists of instances of syntactic entities, which appear within the program under test, mutated by the operator. For example, the mutation operator that replaces a `while` statement by a `do-while` statement has all instances of the `while` statements in its domain. This example, however, illustrates a situation in which the domain is known.

The domain of a mutation operator is the set of all syntactic entities in the program under test which can be mutated by this operator.

Consider a C function having only one declaration statement `int x, y, z`. What kind of syntactic aberrations can one expect in this declaration? One aberration could be that though the programmer intended `z` to be a real variable, it was declared as an integer. Certainly, a mutation operator can be defined to model such an error. However, the list of such aberrations is possibly infinite and, if not impossible, difficult to enumerate. The primary source of this difficulty is the infinite set of type and identifier associations to select from. Thus it becomes difficult to determine the domain for any mutant operator that might operate on a declaration.

The above reasoning leads us to treat declarations as *universe defining* entities in a program. The universe defined by a declaration, such as the one mentioned above, is treated as a collection of facts. Declaration `int x, y, z` states three facts, one for each of the three identifiers. Once we regard declarations to be program entities that state facts, we cannot mutate them because we have assumed that there is no scope for any syntactic aberration. With this reasoning as the basis, declarations in a C program are not mutated. Errors in declarations are expected to manifest through one or more mutants.

Following is the complete list of entities that are not mutated.

- Declarations
- The address operator (&)
- Format strings in input-output functions
- Function declaration headers
- Control line
- Function name indicating a function call. Note that actual parameters in a call are mutated, but the function name is not. This implies that I/O function names such as scanf, printf, open, and so on are not mutated
- Preprocessor conditionals

8.10.2 Linearization

In C, the definition of a statement is recursive. For the purpose of understanding various mutation operators in the statement mutations category, we introduce the concept of *linearization* and *reduced linearized sequence*.

Let S denote a syntactic construct that can be parsed as a C *statement*. Note that *statement* is a syntactic category in C. For an iterative or selection statement denoted by S, ${}^{c}S$ denotes the condition controlling the execution of S. If S is a `for` statement, then ${}^{e}S$ denotes the expression executed immediately after one execution of the loop body and immediately before the next iteration of the loop body, if any, is about to begin. Again, if S is a `for` statement, then ${}^{i}S$ denotes the initialization expression executed exactly once for each execution of S. If the controlling condition is missing, then ${}^{c}S$ defaults to `true`.

Using the above notation, if S is an `if` statement, we shall refer to the execution of S in an execution sequence as ${}^{c}S$. If S denotes a `for` statement, then in an execution sequence we shall refer to the execution of S by one reference to ${}^{i}S$, one or more references to ${}^{c}S$, and zero or more references to ${}^{e}S$. If S is a compound statement, then referring to S in an execution sequence merely refers to any storage allocation activity.

Example 8.22 Consider the following `for` statement:

```
for (m=0, n=0; isdigit(s[i]); i++)
   n =10* n +(s[i]) - '0');
```

Denoting the above `for` statement by S, we get,

${}^{i}S$: m=0, n=0
${}^{c}S$: isdigit(s[i]), and
${}^{e}S$: i++.

If S denotes the following `for` statement,

```
 for (; ;){
⋮
 }
```

then we have,

${}^{i}S$: ; (the null expression-statement),
${}^{c}S$: `true`, and
${}^{e}S$: ; (the null expression-statement).

Let T_f and T_S denote, respectively, the parse trees of function f and statement S. A node of T_S is said to be *identifiable* if it is labeled by any one of the following syntactic categories:

- statement
- labeled statement
- expression statement
- compound statement
- selection statement
- iteration statement
- jump statement

A *linearization* of S is obtained by traversing T_S in preorder and listing, in sequence, only the identifiable nodes of T_S.

A linearization of a statement in C is obtain through a pre-order traversal of its syntax tree.

For any X, let $X_j^i, 1 \leq j \leq i$ denote the sequence $X_j\, X_{j+1} \ldots X_{i-1}\, X_i$. Let $S_L = S_1^l, l \geq 1$ denote the linearization of S. If $S_i\, S_{i+1}$ is a pair of adjacent elements in S_L such that S_{i+1} is the direct descendent of S_i in T_S and there is no other direct descendent of S_i, then $S_i S_{i+1}$ is considered to be a *collapsible* pair with S_i being the *head* of the pair. A *reduced linearized sequence* of S, abbreviated as *RLS*, is obtained by recursively replacing all collapsible elements of S_L by their heads. The *RLS* of a function is obtained by considering the entire body of the function as S and finding the *RLS* of S. The *RLS*, obtained by the above method, will yield a statement sequence in which the indices of the statements are not increasing in steps of 1. We shall always simplify the *RLS* by renumbering its elements, so that for any two adjacent elements $S_i\, S_j$, we have $j = i + 1$.

We shall refer to the *RLS* of a function f and a statement S by $RLS(f)$ and $RLS(S)$, respectively.

8.10.3 Execution sequence

Though most mutation operators are designed to simulate simple faults, the expectation of mutation-based testing is that such operators will eventually reveal one or more errors in the program. In this section, we provide some basic definitions that are useful in understanding such operators and their dynamic effects on the program under test.

When f executes, the elements of $RLS(f)$ will be executed in an order determined by the test case and any path conditions in $RLS(f)$. Let $E(f, t) = S_1^m$, $m \geq 1$ be the execution sequence of $RLS(f) = S_1^n\, R$ for test case t, where S_j, $1 \leq j \leq m - 1$ is any one of S_i, $1 \leq i \leq n$ and S_i is not a `return` statement. We assume that f terminates on t. Thus $S_m = R'$, where R' is R or any other return statement in $RLS(f)$.

Any proper prefix S_1^k, $0 < k < m$ of $E(f,t)$, where $S_k = R'$, is a *prematurely terminating execution sequence* (subsequently referred to as *PTES* for brevity) and is denoted by $E^p(f,t)$. S_{k+1}^m is known as the *suffix* of $E(f,t)$ and is denoted by $E^s(f,t)$; $E^l(f,t)$ denotes the last statement of the execution sequence of f. If f is terminating, $E^l(f,t)$=`return`.

Let $E_1 = S_i^j$ and $E_2 = Q_k^l$ be two execution sequences. We say that E_1 and E_2 are *identical* if and only if $i = k, j = l$, and $S_q = Q_q$, $i \le q \le j$. As a simple example, if f and f' consist of one assignment each, namely, $a = b + c$ and $a = b - c$, respectively, then there is no t for which $E(f,t)$ and $E(f',t)$ are identical. It must be noted that the output generated by two execution sequences may be the same even though the sequences are not identical. In the above example, for any test case t that has $c = 0$, $P_f(t) = P_{f'}(t)$.

Example 8.23 Consider the function trim defined below. Note that this and several other illustrative examples in this chapter borrow program fragments for mutation from the well-known book by Brian Kernighan and Dennis Ritchie titled "The C programming Language."

/* This function is from p 65 of Kernighan and Ritchie's book. */

```
      int trim(char s[])
S1    {
        int n;                                                  F1
S2      for (n = strlen(s)-1; n >= 0; n--)
S3        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
S4          break;
S5      s[n+1] = '\0';
S6      return n;
      }
```

We have $RLS(trim) = S_1 S_2 S_3 S_4 S_5 S_6$. Let the test case t be such that the input parameter s evaluates to *ab* (a space character follows b), then the execution sequence $E(trim,t)$ is: $S_1{}^i S_2{}^c S_2{}^c S_3{}^c S_2{}^c S_3 S_4 S_5 S_6$. $S_1{}^i S_2$ is one prefix of $E(f,t)$ and $S_4 S_5 S_6$ is one suffix of $E(trim,t)$. Note that there are several other prefixes and suffixes of $E(trim,t)$. $S_1{}^i S_2{}^c S_2 S_6$ is a proper prefix of $E(f,t)$.

The reduced linearized sequence of a statement aids in determining the domain of mutation operators that have statements in their domain.

Analogous to the execution sequence for $RLS(f)$, we define the execution sequence of $RLS(S)$ denoted by $E(S,t)$ with $E^p(S,t)$, $E^s(S,t)$, and $E^l(S,t)$ corresponding to the usual interpretation.

The composition of two execution sequences $E_1 = p_1^k$ and $E_2 = q_1^l$ is $p_1^k q_1^l$ and is written as $E_1 \circ E_2$. The conditional composition of E_1 and E_2 with respect to condition c is written as $E_1 |_c E_2$. It is defined as:

$$E_1 \mid_c E_2 = \begin{cases} E_1 & \text{if } c \text{ is false,} \\ E_1 \circ E_2 & \text{otherwise.} \end{cases}$$

In the above definition, condition c is assumed to be evaluated after the entire E_1 has been executed. Note that $\circ$ has the same effect as $\mid_{\texttt{true}}$. $\circ$ associates from left to right and $\mid_c$ associates from right to left. Thus, we have:

$$E_1 \circ E_2 \circ E_3 = (E_1 \circ E_2) \circ E_3$$
$$E_1 \mid_{c_1} E_2 \mid_{c_2} E_3 = E_1 \mid_{c_1} (E_2 \mid_{c_2} E_3)$$

$E(f,*)$ ($E(S,*)$) denotes the execution sequence of function f (statement S) on the *current* values of all the variables used by f (S). We shall use this notation while defining execution sequences of C functions and statements.

Let S, S_1, and S_2 denote a C statement other than `break`, `continue`, `goto`, and `switch`, unless specified otherwise. The following rules can be used to determine execution sequences for any C function.

R 1 E({ },t) is the null sequence.

R 2 $E(\{\},t) \circ E(S,t) = E(S,t) = E(S,t) \circ E(\{\})$

R 3 $E(\{\},t) \mid_c E(S,t) = \mid_c E(S,t) = \begin{cases} \text{null sequence} & \text{if } c \text{ is false,} \\ E(S,t) & \text{otherwise.} \end{cases}$

R 4 If S is an *assignment-expression*, then $E(S,t) = S$.

R 5 For any statement S, $E(S,t) = RLS(S)$, if $RLS(S)$ contains no statements other than zero or more assignment-expressions. If $RLS(S)$ contains any statement other than the assignment-expression, the above equality is not guaranteed due to the possible presence of conditional and iterative statements.

R 6 If $S = S_1\,;\,S_2$; then $E(S,t) = E(S_1,t) \circ E(S_2,*)$.

R 7 If $S = \texttt{while}$ (c) S_1', then

$$E(S,t) = \mid_c (E(S_1',*) \circ E(S,*)).$$

If $RLS(S) = S_1^n$, $n > 1$, and $S_i = \texttt{continue}$, $1 \le i \le n$, then

$$E(S,t) = \mid_c E(S_1^i,*) \circ (\mid_{(E^l(S_1^i,*) \neq \texttt{continue})} E(S_{i+1}^n,*)) \circ E(S,*).$$

If $RLS(S) = S_1^n$, $n > 1$, and $S_i = \texttt{break}$, $1 \le i \le n$, then

$$E(S,t) = \mid_c E(S_1^i,*) \circ (\mid_{(E^l(S_1^i,*) \neq \texttt{break})} (E(S_{i+1}^n,*) \circ E(S,*))).$$

R 8 If $S =$ `do` S_1`while`(c); then

$$E(S,t) = E(S_1,t) \mid_c E(S,*).$$

If $RLS(S)$ contains a `continue`, or a `break`, then its execution sequence can be derived using the method indicated for the `while` statement.

R 9 If $S =$ `if` (c) S_1, then

$$E(S,t) = \mid_c E(S_1,*).$$

R 10 If $S =$ `if` (c) S_1 `else` : S_2, then

$$E(S,t) = \begin{cases} E(S_1,t) & \text{if } c \text{ is true,} \\ E(S_2,t) & \text{otherwise.} \end{cases}$$

Example 8.24 Consider S_3, the `if` statement, in F1. We have $RLS(S_3) = S_3\ S_4$. Assuming the test case of the Example 8.23 and $n = 3$, we get $E(S_3,*) = {}^cS_3$. If $n = 2$, then $E(S_3,*) = {}^cS_3\ S_4$. Similarly, for S_2, which is a `for` statement, we get $E(S_2,*) = {}^iS_2\ {}^cS_2\ {}^cS_3\ {}^cS_2\ {}^cS_3\ S_4$. For the entire function body, we get $E(trim,t) = S_1\ E(S_2,*) \circ E(S_5,*) \circ E(S_6,*)$.

8.10.4 Effect of an execution sequence

As before, let P denote the program under test, f a function in P, that is to be mutated, and t a test case. Assuming that P terminates, let $P_f(t)$ denote the output generated by executing P on t. The subscript f with P is to emphasize the fact that it is the function f that is being mutated.

The execution sequence of a function in program P under test must be different from that of its mutant. However, this is only a necessary condition for the corresponding mutant of P to be distinguished from its parent.

We say that $E(f,*)$ has a distinguishable effect on the output of P, if $P_f(t) \neq P_{f'}(t)$, where f' is a mutant of f. We consider $E(f,*)$ to be a *distinguishing execution sequence* (DES) of $P_f(t)$ with respect to f'.

Given f and its mutant f', for a test case t to distinguish f', it is necessary, but not sufficient, that $E(f,t)$ be different from $E(f',t)$. The sufficiency condition is that $P_f(t) \neq P_{f'}(t)$ implying that $E(f,t)$ is a DES for $P_f(t)$ with respect to f'.

While describing the mutant operators, we shall often use DES to indicate when a test case is sufficient to distinguish between a program and its mutant. Examining the execution sequences of a function, or a statement, can be useful in constructing a test case that distinguishes a mutant.

Example 8.25 To illustrate the notion of the effect of an execution sequence, consider the function *trim* defined in F1. Suppose that the output of interest is the string denoted by s. If the test case t is such that s consists of the three characters a, b, and space, in that order, then $E(trim, t)$ generates the string *ab* as the output. As this is the intended output, we consider it to be correct.

Now suppose that we modify *trim* by mutating S_4 in F1 to `continue`. Denoting the modified function by $trim'$, we get

$$E(trim', t) = S_1\, {}^{i}S_2\, {}^{c}S_2\, {}^{c}S_3\, {}^{e}S_2\, {}^{c}S_3\, {}^{e}S_2\, {}^{c}S_3\, S_4\, {}^{e}S_2\, {}^{c}S_3\, S_4\, {}^{e}S_2\, S_5\, S_6).$$

The output generated due to $E(trim', t)$ is different from that generated due to $E(trim, t)$. Thus, $E(trim, t)$ is a DES for $P_{trim}(t)$ with respect to the function $trim'$.

DESs are essential to kill mutants. To obtain a DES for function f, a suitable test case t needs to be constructed such that $E(f, t)$ is a DES for $P_f(t)$ with respect to f'.

8.10.5 Global and local identifier sets

For defining variable mutations in Section 8.10.10, we need the concept of global and local sets, defined in this section, and global and local reference sets, defined in the next section.

Let f denote a C function to be mutated. An identifier denoting a variable, that can be used inside f, but is not declared in f, is considered global to f. Let G_f denote the set of all such global identifiers for f. Note that any *external* identifier is in G_f unless it is also declared in f. While computing G_f, it is assumed that all files specified in one or more `# include` control lines have been *included* by the C preprocessor. Thus any global declaration within the files listed in a `# include` also contributes to G_f.

A variable used inside a function f but not declared in it is considered global to f. A variable declared inside f is considered local to f. This gives rise to set so global and local variables in function f.

Let L_f denote the set of all identifiers declared either as parameters of f or at the head of its body. Identifiers denoting functions do not belong to G_f or L_f.

In C, it is possible for a function f to have nested compound statements such that an inner compound statement S has declarations at its head. In such a situation, the global and local sets for S can be computed using the scope rules in C.

We define GS_f, GP_f, GT_f, and GA_f as subsets of G_f which consist of, respectively, identifiers declared as scalars, pointers to an entity, structures, and arrays. Note that these four subsets are pairwise disjoint. Similarly, we define LS_f, LP_f, LT_f, and LA_f as the pairwise disjoint subsets of L_f.

8.10.6 Global and local reference sets

A global reference set for function f is the set of all variables that are referenced in f but not declared in f. Variables that are referenced inside f and also declared inside f constitute a local reference set.

Use of an identifier within an expression is considered a *reference*. In general, a reference can be *multilevel* implying that it can be composed of one or more subreferences. Thus, for example, if *ps* is a pointer to a structure with components *a* and *b*, then in $(*ps).a$, *ps* is a reference and $*ps$ and $(*ps).a$ are two subreferences. Further, $*ps.a$ is a 3-level reference. At level 1, we have *ps*, at level 2 we have $(*ps)$, and finally at level 3 we have $(*ps).a$. Note that in C, $(*ps).a$ has the same meaning as $ps->a$.

The global and local reference sets consist of references at level 2 or higher. Any references at level 1 are in the global and local sets defined earlier. We shall use GR_f and LR_f to denote, respectively, the global and local reference sets for function f.

Referencing a component of an array or a structure may yield a scalar quantity. Similarly, dereferencing a pointer may also yield a scalar quantity. All such references are known as *scalar* references. Let GRS_f and LRS_f denote sets of all such global and local scalar references, respectively. If a reference is constructed from an element declared in the global scope of f, then it is a global reference, otherwise it is a local reference.

We now define GS'_f and LS'_f by augmenting GS_f and LS_f as follows:

$$GS'_f = GRS_f \cup GS_f$$

$$LS'_f = LRS_f \cup LS_f$$

GS'_f and LS'_f are termed as scalar global and local reference sets for function f, respectively.

Similarly, we define array, pointer, and structure reference sets denoted by GRA_f, GRP_f, GRT_f, LRA_f, LRP_f, and LRT_f. Using these, we can construct the augmented global and local sets GA'_f, GP'_f, GT'_f, LA'_f, LP'_f, and LT'_f.

For example, if an array is a member of a structure, then a reference to this member is an array reference and hence belongs to the array reference set. Similarly, if a structure is an element of an array, then a reference to an element of this array is a structure reference and hence belongs to the structure reference set.

On an initial examination, our definition of global and local reference sets might appear to be ambiguous especially with respect to a pointer to an *entity*. An *entity* in the present context can be a scalar, an array, a structure, or a pointer. Function references are not mutated. However, if *fp* is a pointer to some entity, then *fp* is in set GRP_f or LRP_f depending on its place of declaration. On the other hand, if *fp* is an entity of pointer(s), then it is in any one of the sets GRX_f or LRX_f where X could be any one of the letters A, P, or T.

Example 8.26 To illustrate our definitions, consider the following external declarations for function f.

```
int i, j; char c, d; double r, s;
int *p, *q [ 3 ];
struct point {
   int x;
   int y;
};

struct rect {
   struct point p1;
   struct point p2;                                    F2
};

struct rect screen;
 struct key {
   char * word;
   int count;
} keytab [ NKEYS ];
```

The global sets corresponding to the above declarations are

$$G_f = \{i, j, c, d, r, s, p, q, screen, keytab\},$$
$$GS_f = \{i, j, c, d, r, s\},$$
$$GP_f = \{p\},$$
$$GT_f = \{screen\}, \text{and}$$
$$GA_f = \{q, keytab\}.$$

Note that structure components x, y, *word*, and *count* do not belong to any global set. Type names, such as *rect* and *key* above, are not in any global set. Further, type names do not participate in mutation due to reasons outlined in Section 8.10.1.

Now, suppose that the following declarations are within function f.

```
int fi; double fx; int *fp; int (*fpa) (20)
struct rect fr; struct rect *fprct;
int fa [10 ]; char *fname [ nchar ]
```

Then the local sets for f are

$$
\begin{aligned}
L_f &= \{fi, fx, fp, fpa, fr, fprct, fa, fname\},\\
LA_f &= \{fa, fname\},\\
LP_f &= \{fp, fpa, fprct\},\\
LS_f &= \{fi, fx\}, \text{and}\\
LT_f &= \{fr\}.
\end{aligned}
$$

To illustrate reference sets, suppose that f contains the following references (the specific statement context in which these references are made is of no concern for the moment).

```
i * j + fi
r + s – fx + fa [ i ]
*p += 1
*q [j] = *p
screen . p1 = screen . p2
screen . p1 . x = i
keytab [j] . count = *p
p = q [i]
fr = screen
*fname [j] = keytab [ i ] . word
fprct = &screen
```

The global and local reference sets corresponding to the above references are

$$
\begin{aligned}
GRA_f &= \{\ \}\\
GRP_f &= \{q[i], keytab[i].word, \&screen\},\\
GRS_f &= \{keytab[j].count, *p, *q[j], screen.p1.x\},\\
GRT_f &= \{keytab[i], keytab[j], screen.p1, screen.p2\},\\
LRA_f &= \{\ \},\\
LRP_f &= \{fname[j]\},\\
LRS_f &= \{*fname[j], fa[i]\}, \text{and}\\
LRT_f &= \{\ \}.
\end{aligned}
$$

The above sets can be used to augment the local sets.

Analogous to the global and local sets of variables, we define global and local sets of constants: GC_f and LC_f. GC_f is the set of all constants global to f. LC_f is the set of all constants local to f. Note that a constant can be used within a declaration or in an expression.

We define GCI_f, GCR_f, GCC_f, and GCP_f to be subsets of GC_f consisting of only integer, real, character, and pointer constants. GCP_f consists of only `null`. LCI_f, LCR_f, LCC_f, and LCP_f are defined similarly.

8.10.7 Mutating program constants

We begin our introduction to the mutation operators for C with operators that mutate constants. These operators model coincidental correctness and, in this sense, are similar to scalar variable replacement operators discussed later in Section 8.10.10. A complete list of such operators is available in Table 8.2.

An incorrect use of constants in a program is modeled by the CGCR, CLSR, CGSR, CRCR, and CLCR operators.

Required constant replacement

Let I and R denote, respectively, the sets $\{0, 1, -1, u_i\}$ and $\{0.0, 1.0, -1.0, u_r\}$. u_i and u_r denote user specified integer and real constants, respectively. Use of a variable, where an element of I or R was the correct choice, is the fault modeled by **CRCR**.

Each scalar reference is replaced systematically by elements of I or R. If the scalar reference is integral, I is used. For references that are of type floating, R is used. Reference to an entity via a pointer is replaced by `null`. Left operands of the assignment operators as well as the ++ and −− operators are not mutated.

Table 8.2 Mutation operators for the C programming language that mutate program constants.

Mutop	Domain	Description
CGCR	Constants	Constant replacement using global constants
CLSR	Constants	Constant for scalar replacement using local constants
CGSR	Constants	Constant for scalar replacement using global constants
CRCR	Constants	Required constant replacement
CLCR	Constants	Constant replacement using local constants

Example 8.27 Consider the statement $k = j + *p$, where k and j are integers and p is a pointer to an integer. When applied to the above statement, the **CRCR** mutation operator generates the following mutants.

```
k = 0 + * p
k = 1 + * p
k = -1 + * p                                    M1
k = u_i + *p
k = j +null
```

A **CRCR** mutant encourages a tester to design at least one test case that forces the variable replaced to take on values other than from the set I or R. Thus such a mutant attempts to overcome coincidental correctness of P.

Constant for Constant Replacement

Just as a programmer may mistakenly use one identifier for another, a possibility exists that one may use a constant for another. Mutation operators **CGCR** and **CLCR** model such faults. These two operators mutate constants in f using, respectively, the sets GC_f and LC_f.

Example 8.28 Suppose that constant 5 appears in an expression, and $GC_f = \{0, 1.99, 'c'\}$, then 5 will be mutated to 0, 1.99, and $'c'$ thereby producing three mutants.

Null is not mutated.

Pointer constant, `null`, is not mutated. Left operands of assignment, $++$ and $--$ operators are also not mutated.

Constant for Scalar Replacement

Use of a scalar variable, instead of a constant, is the fault modeled by mutation operators **CGSR** and **CLSR**. **CGSR** mutates all occurrences of scalar variables or scalar references by constants from the set GC_f. **CLSR** is similar to **CGSR** except that it uses LC_f for mutation. Left operands of assignment, $++$ and $--$ operators are not mutated.

"Mutating operators" refers the act of mutating operators in a C program and is different from "mutation operators."

8.10.8 Mutating operators

Mutation operators in this category model common errors made while using various operators in C. Do not be confused with the overloaded use of the term "operator." We use the term "mutation operator" to refer to

a mutation operator as discussed earlier, and the term "operator" to refer to the operators in C such as the arithmetic operator + or the relational operator <.

8.10.9 Binary operator mutations

The incorrect choice of a binary C-operator within an expression is the fault modeled by this mutation operator. The binary mutation operators fall into two categories: *Comparable Operator Replacement* (**Ocor**) and *Incomparable Operator Replacement* (**Oior**). Within each subcategory, the mutation operators correspond to either the *non-assignment* or to the *assignment* operators in C. Tables 8.3, 8.4, and 8.5 list all binary mutation operators in C.

The binary operator mutation operators model the errors due to an incorrect use of binary operators such as + or /.

Each binary mutation operator systematically *replaces* a C operator in its domain by operators in its range. The domain and range for all mutation operators in this category are specified in Tables 8.3, 8.4, and 8.5. In certain contexts, only a subset of arithmetic operators is used. For example, it is illegal to add two pointers, though a pointer may be subtracted from another. All mutation operators that mutate C-operators, are assumed to recognize such exceptional cases to retain the syntactic validity of the mutant.

Table 8.3 Domain and range of mutation operators in **Ocor.**

Name	Domain	Range	Example
OAAA	Arithmetic assignment	Arithmetic assignment	a+= b→ a − = b
OAAN	Arithmetic	Arithmetic	a +b→ a * b
OBBA	Bitwise assignment	Bitwise assignment	a &= b→ a|/= b
OBBN	Bitwise	Bitwise	a & b→ a|/ b
OLLN	Logical	Logical	a && b→ a ‖ b
ORRN	Relational	Relational	a < b→ a <= b
OSSA	Shift assignment	Shift assignment	a <<= b→ a >>= b
OSSN	Shift	Shift	a << b→ a>> b

† Read X → Y as "X gets mutated to Y."

Table 8.4 Domain and range of mutation operators in **Oior**: arithmetic and bitwise.

Name	Domain	Range	Example
OABA	Arithmetic assignment	Bitwise assignment	a += b→ a |/= b
OAEA	Arithmetic assignment	Plain assignment	a += b→ a = b
OABN	Arithmetic	Bitwise	a + b→ a & b
OALN	Arithmetic	Logical	a + b→ a && b
OARN	Arithmetic	Relational	a + b→ a < b
OASA	Arithmetic assignment	Shift assignment	a += b→ a <<= b
OASN	Arithmetic	Shift	a + b→ a << b
OBAA	Bitwise assignment	Arithmetic assignment	a |/= b→ a += b
OBAN	Bitwise	Arithmetic	a & b→ a + b
OBEA	Bitwise assignment	Plain assignment	a &= b→ a =b
OBLN	Bitwise	Logical	a & b→ a && b
OBRN	Bitwise	Relational	a & b→ a < b
OBSA	Bitwise assignment	Shift assignment	a &= b→ a <<= b
OBSN	Bitwise	Shift	a & b→ a << b

† Read X → Y as "X gets mutated to Y."

Unary operator mutations

Mutations in this subcategory consist of mutation operators that model faults in the use of unary operators and conditions. Operators in this category fall into the five subcategories described in the following.

Increment/Decrement: The ++ and −− operators are used frequently in C programs. The **OPPO** and **OMMO** mutation operators model the faults that arise from the incorrect use of these C operators. The incorrect uses modeled are: (a) ++ (or −−) used instead of −− (or ++) and (b) prefix increment (decrement) used instead of postfix increment (decrement).

The **OPPO** operator generates two mutants. An expression such as ++x is mutated to x++ and −−x. An expression, such as x++, will be mutated

Table 8.5 Domain and Range of mutation operators in **Oior**: plain, logical, and relational.

Name	Domain	Range	Example
OEAA	Plain assignment	Arithmetic assignment	a = b→ a += b
OEBA	Plain assignment	Bitwise assignment	a =b→ a &= b
OESA	Plain assignment	Shift assignment	a = b→ a <<= b
OLAN	Logical	Arithmetic	a && b→ a + b
OLBN	Logical	Bitwise	a && b→ a & b
OLRN	Logical	Relational	a && b→ a < b
OLSN	Logical	Shift	a && b→ a << b
ORAN	Relational	Arithmetic	a < b→ a + b
ORBN	Relational	Bitwise	a < b→ a & b
ORLN	Relational	Logical	a < b→ a && b
ORSN	Relational	Shift	a < b→ a << b
OSAA	Shift assignment	Arithmetic assignment	a <<= b→ a += b
OSAN	Shift	Arithmetic	a<< b→ a + b
OSBA	Shift assignment	Bitwise assignment	a << b→ a |/= b
OSBN	Shift	Bitwise	a << b→ a & b
OSEA	Shift assignment	Plain assignment	a <<= b→ a = b
OSLN	Shift	Logical	a << b→ a && b
OSRN	Shift	Relational	a << b→ a < b

† Read X → Y as 'X gets mutated to Y'.

to ++x and x−−. The **OMMO** operator behaves similarly. It mutates −−x to x−− and ++x. It also mutates x−− to −−x and x++. Both the operators will not mutate an expression if its value is not used. For example, an expression such as i++ in a `for` header will not be mutated, thereby avoiding the creation of an equivalent mutant. An expression such as *x++ will be mutated to *++x and *x−−.

Logical Negation: Often, the sense of the condition used in iterative and selective statements is reversed. **OLNG** models this fault. Consider the

expression *x op y*, where *op* can be any one of the two logical operators: && and ||. **OLNG** will generate three mutants of such an expression as follows: *x op* !*y*, !*x op y*, and !(*x op y*).

Logical Context Negation: In selective and iterative statements, excluding the `switch`, often the sense of the controlling condition is reversed. **OCNG** models this fault. The controlling condition in the iterative and selection statements is negated. The following examples illustrate how OCNG mutates expressions in iterative and selective statements.

```
if (expression) statement →
  if (!expression) statement

if (expression) statement else statement →
  if (!expression) statement else statement

while (expression) statement →
  while (!expression) statement

do statement while (expression) →
  do statement while (!expression)

for (expression; expression; expression) statement →
  for (expression; !expression, expression)statement

expression ? expression : conditional expression →
  !expression ? expression : conditional expression
```

When applied on an iteration statement, application of the **OCNG** operator may generate mutants with infinite loops. Further, this application may also generate mutants generated by **OLNG**. Note that a condition such as (x<y) in an `if` statement will not be mutated by **OLNG**. However, the condition ((x<y&&p>q) will be mutated by both **OLNG** and **OCNG** to (!(x<y)&&(p>q)).

Bitwise Negation: The sense of the bitwise expressions may often be reversed. Thus, instead of using (or not using) the one's complement operator, the programmer may not use (or may use) the bitwise negation operator. The **OBNG** operator models this fault.

Consider an expression of the form *x op y*, where *op* is one of the bitwise operators: |/ and &. The **OBNG** operator mutates this expression to: *x op*~ *y*,~*x op y*, and~(*x op y*). **OBNG** does not consider the iterative and conditional operators as special cases. Thus, for example, a statement such

as `if` (x && a |/ b) p = q will get mutated to the following statements by **OBNG**:

```
if (x && a |/~ b) p = q
if (x &&~ a |/ b) p = q
if (x &&~ (a |/ b)) p = q
```

Indirection Operator Precedence Mutation: Expressions constructed using a combination of ++, −−, and the indirection operator (*), can often contain precedence faults. For example, using *p++ when (*p)++ was meant, is one such fault. **OIPM** operator models such faults.

OIPM mutates a reference of the form $*x\ op$ to $(*x)\ op$ and $op\ (*x)$, where op can be ++ and −−. Recall that in C, $*x\ op$ implies $*(x\ op)$. If op is of the form [y], then only $(*x)\ op$ is generated. For example, a reference such as $*x[p]$ will be mutated to (*x)[p].

The above definition is for the case when only one indirection operator has been used to form the reference. In general, there could be several indirection operators used in formulating a reference. For example, if x is declared as `int` *** x, then *** x ++ is a valid reference in C. A more general definition of **OIPM** takes care of this case.

Consider the following reference $\underbrace{**\ldots*}_{n}\ x\ op$. **OIPM** systematically mutates this reference to the following references:

$$\underbrace{**\ldots*}_{n-1} \quad (*x) \quad op$$

$$\underbrace{**\ldots*}_{n-1} \quad op \quad (*x)$$

$$\underbrace{**\ldots*}_{n-2} \quad (**x) \quad op$$

$$\underbrace{**\ldots*}_{n-2} \quad op \quad (**x)$$

$$\vdots \qquad \vdots \qquad \vdots$$

$$*(\underbrace{**\ldots*}_{n-1}x)op$$

$$* \quad op \quad (\underbrace{**\ldots*}_{n-1}x)$$

$$(\underbrace{**\ldots*}_{n}x)op$$

$$op \quad (\underbrace{**\ldots*}_{n}x)$$

Multiple indirection operators are used infrequently. Hence, in most cases, we expect **OIPM** to generate two mutants for each reference involving the indirection operator.

Cast Operator Replacement: A cast operator, referred to as *cast*, is used to explicitly indicate the type of an operand. Faults in such usage are modeled by **OCOR**.

Every occurrence of a cast operator is mutated by **OCOR**. Casts are mutated in accordance with the restrictions listed below. These restrictions are derived from the rules of C as specified in the ANSI C standard. While reading the cast mutations described below, ↔ may be read as "gets mutated to." All entities to the left of ↔ get mutated to the entities on its right and vice versa. The notation X* can be read as "X and all mutations of X excluding duplicates."

```
char ↔ signed char  unsigned char
    int*  float*

int ↔ signed int  unsigned int
    short int  long int
    signed long int  signed long int
    float*  char*

float ↔ double  long double
    int*  char*

double ↔ char*  int*
    float*
```

Example 8.29 Consider the statement:

`return (unsigned int) (next/65536) % 32768`

Sample mutants generated when **OCOR** is applied to the above statement are shown below (only cast mutations are listed).

```
short int  long int
float double
```

Note that the cast operators, other than those described in this section, are not mutated. For example, the casts in the following statement are not mutated:

qsort((`void` **) lineptr, 0, nlines-1, (`int`(*) (`void`*, `void`*))(numeric ? numcmp :strcmp))

Table 8.6 List of mutation operators for statements in C.

Mutop	Domain	Description
SBRC	`break`	`break` replacement by `continue`
SBRn	`break`	Break out to n^{th} level
SCRB	`continue`	`continue` replacement by `break`
SDWD	`do-while`	`do-while` replacement by `while`
SGLR	`goto`	`goto` label replacement
SMVB	Statement	Move brace up and down
SRSR	`return`	`return` replacement
SSDL	Statement	Statement deletion
SSOM	Statement	Sequence Operator Mutation
STRI	`if` Statement	Trap on `if` condition
STRP	Statement	Trap on statement execution
SMTC	Iterative statements	*n*-trip continue
SSWM	`switch` statement	`switch` statement mutation
SMTT	Iterative statement	*n*–trip trap
SWDD	`while`	`while` replacement by `do-while`

The decision not to mutate certain casts was motivated by their infrequent use and the low probability of a fault that could be modeled by mutation. For example, a cast such as `void` ** is used infrequently and when used, the chances of it being mistaken for, say, an `int`, appear to be low.

8.10.10 Mutating statements

We now describe each one of the mutation operators that mutate entire statements or their key syntactic elements. A complete list of such operators is given in Table 8.6. For each mutation operator, its definition and the fault modeled is provided. The domain of a mutation operator is described in terms of the effected syntactic entity.

Statement mutation operators model faults due to errors in the placement or construction of statements.

Recall that some statement mutation operators are included to ensure code coverage and do not model any specific fault. The STRP is one such operator.

The operator and variable mutations described in subsequent sections, also effect statements. However, they are not intended to model faults in the explicit composition of the *selection, iteration*, and *jump* statements.

Trap on Statement Execution

This operator is intended to reveal unreachable code in the program. Each statement is systematically replaced by trap_on_statement(). Mutant execution terminates when trap_on_statement is executed. The mutant is considered distinguished.

Example 8.30 Consider the following program fragment:

```
while (x != y)
 {
  if (x < y)                                    F3
   y -= x;
  else
   x -= y;
 }
```

Application of **STRP** to the above statement generates a total of four mutants shown in M2, M3, M4, and M5. Test cases that distinguish all these four mutants are sufficient to guarantee that all four statements in F3 have been executed at least once.

```
trap_on_statement();                            M2
while (x != y)
 {
  trap_on_statement();                          M3
 }
while (x != y)
 {
  if (x < y)                                    M4
   trap_on_statement();
  else
   x - = y;
 }
while (x != y)
 {
  if (x < y)                                    M5
   y -= x;
  else
   trap_on_statement();
 }
```

If **STRP** is used with the RAP set to include the entire program, the tester will be forced to design test cases that guarantee that all statements have been executed. Failure to design such a test set implies that there is some unreachable code in the program.

Trap on if *Condition*

The **STRI** mutation operator is designed to provide branch analysis for any `if`-statements in *P*. When used in addition to the **STRP**, **SSWM**, and **SMTT** operators, complete branch analysis can be performed. **STRI** generates two mutants for each `if` statement.

Example 8.31 The two mutants generated for statement `if` (*e*)*S* are as follows:

```
v = e
if (trap_on_true(v)) S                    M6
v = e
if (trap_on_false(v)) S                   M7
```

Here *v* is assumed to be a new scalar identifier not declared in *P*. The type of *v* is the same as that of *e*.

When trap_on_true (trap_on_false) is executed, the mutant is distinguished if the function argument value is true (false). If the argument value is not true (false), then the function returns false (true) and the mutant execution continues.

The **STRI** operator encourages the tester to generate test cases so that each branch specified by an `if` statement in *P*, is exercised at least once. For an implementor of a mutation-based tool, it is useful to note that **STRI** provides partial branch analysis for `if` statements. For example, consider a statement of the form: `if` (c) S_1 `else` S_2. The **STRI** operator will have this statement replaced by the following statements to generate two mutants.

- `if` (c) trap_on_statement() `else` S_2
- `if` (c) S_1 `else` trap_on_statement()

Distinguishing both these mutants implies that both the branches of the `if-else` statement have been traversed. However, when used with a `if` statement without an *else* clause, **STRI** may fail to provide coverage of both the branches.

A mutant obtained by deleting a statement forces the tester to generate a test that demonstrates an impact of the deleted statement on the output of the program under test.

Statement Deletion

SSDL is designed to show that each statement in *P* has an effect on the output. SSDL encourages the tester to design a test set that causes all statements in the RAP to be executed and generates outputs that are different from the program under test. When applied on *P*, **SSDL** systematically deletes each statement in *RLS*(*f*).

Example 8.32 When **SSDL** is applied to F3, four mutants are generated as shown in M8, M9, M10, and M11.

```
;                                                    M8
while (x != y)
 {                                                   M9
 }
while (x != y)
 {
  if (x < y)                                         M10
   ;
  else
   x -= y;
 }
while (x != y)
 {
  if (x < y)                                         M11
   y -= x;
  else
   ;
 }
```

To maintain the syntactic validity of the mutant, **SSDL** ensures that the semicolons are retained when a statement is deleted. In accordance with the syntax of C, the semicolon appears only at the end of (i) *expression-statement* and (ii) `do-while` *iteration-statement*. Thus, while mutating an *expression-statement*, **SSDL** deletes the optional *expression* from the statement, retaining the semicolon. Similarly, while mutating a `do-while` *iteration-statement*, the semicolon that terminates this statement is retained. In other cases, such as the *selection-statement*, the semicolon automatically gets retained as it is not a part of the syntactic entity being mutated.

Replacing a statement by a return forces a tester to generate a test that demonstrates the effect of all the statements that following the replaced statement on the output of the program under test.

`return` *Statement Replacement*

When a function f executes on test case t, it is possible that due to some fault in the composition of f, certain suffixes of $E(f, t)$ do not affect the output of P. In other words, a suffix may not be a DES of $P_f(t)$ with respect to f' obtained by replacing an element of $RLS(f)$ by a `return`. The **SRSR** operator models such faults.

If $E(f, t) = s_1^m R$, then there are $m + 1$ possible suffixes of $E(f, t)$. These are enumerated below:

$$
\begin{aligned}
s_1\, s_2 \ldots s_{m-1}\, s_m\, R \\
s_2 \ldots s_{m-1}\, s_m\, R \\
s_{m-1}\, s_m\, R \\
\vdots \\
R
\end{aligned}
$$

In case f consists of loops, m could be made arbitrarily large by manipulating the test cases. The **SRSR** operator creates mutants that generate a subset of all possible *PMES*'s of $E(f, t)$.

Let $R_1, R_2, \ldots, R_k$ be the k `return` statements in f. If there is no such statement, a parameterless `return` is assumed to be placed at the end of the text of f. Thus, for our purpose, $k \geq 1$. The **SRSR** operator will systematically replace each statement in $RLS(f)$ by each one of the k `return` statements. The **SRSR** operator encourages the tester to generate at least one test case that ensures that $E^s(f, t)$ is a DES for the program under test.

Example 8.33 Consider the following function definition:

```
/* This is an example from p 69 of Kernighan and Ritichie's book.*/
int strindex(char s[], char t[])
{
  int i, j, k ;
  for (i = 0; s[i] != '\0'; i++){
   for (j=i; k=0; t[k] != '\0' && s[j] == t[k] ; j++ ; k++)
    ;                                                          F4
   if (k>0 && t[k] == '\0')
    return i;
  }
  return −1;
}
```

A total of six mutants are generated when the SRSR operator is applied to `strindex`., two of which are M12 and M13.

```
int strindex(char s[], char t[])
{
  int i, j, k ;
/* The outer for statement replaced by return i.
  return i;  ← /* Mutated statement. */                          M12
  return −1;
}
```

```
/* This mutant has been obtained by replacing
   the inner for by return −1.
int strindex(char s[ ], char t[ ])
{
   int i, j, k ;
   for (i = 0; s[i] != '/ 0'; i++){
    return −1;   ←  /* Mutated statement. */                M13
     if (k>0 && t[k] == ' 0')
     return i;
   }
   return −1;
}
```

Note that both M12 and M13 generate the shortest possible *PMES* for f.

goto *Label Replacement*

The replacement of a label in a jump statement models an erroneous jump destination.

In some function f, the destination of a `goto` may be incorrect. Altering this destination is expected to generate an execution sequence different from $E(f, t)$. Suppose that `goto` L and `goto` M are two `goto` statements in f. We say that these are two `goto` statements are distinct if L and M are different labels. Let `goto` l_1, `goto` $l_2, \ldots,$ `goto` l_n be n distinct `goto` statements in f. The **SGLR** operator systematically mutates label l_i in `goto` l_i to $(n-1)$ labels $l_1, l_2, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n$. If $n = 1$, no mutants are generated by **SGLR**.

`continue` *Replacement by* `break`

The incorrect use of the continue and break statements is modeled by a few mutation operators.

A `continue` statement terminates the *current* iteration of the immediately surrounding loop and initiates the *next* iteration. Instead of the `continue`, the programmer might have intended a `break` that forces the loop to terminate. This is one fault modeled by **SCRB**. Incorrect placement of `continue` is another fault that **SCRB** expects to reveal. **SCRB** replaces the `continue` statement by `break`.

Given S that denotes the innermost loop that contains the `continue` statement, the **SCRB** operator encourages the tester to construct a test case t to show that $E(S, *)$ is a DES for $P_S(t)$ with respect to the mutated S.

`break` *Replacement by* `continue`

Using `break` instead of a `continue` or misplacing a `break` are the two faults modeled by **SBRC**. The `break` statement is replaced by `continue`.

If S denotes the innermost loop containing the `break` statement, then SBRC encourages the tester to construct a test case t to show that $E(S, t)$ is a DES for $P_S(t)$ with respect to S', where S' is a mutant of S.

Break Out to nth Enclosing Level

Execution of a `break` inside a loop forces the loop to terminate. This causes the resumption of execution of the outer loop, if any. However, the condition that caused the execution of `break` might be intended to terminate the execution of the immediately enclosing loop, or in general, the nth enclosing loop. This is the fault modeled by **SBRn**.

Let a `break` (or a `continue`) statement be inside a loop nested n levels deep. A statement with only one enclosing loop is considered to be nested one level deep. The **SBRn** operator systematically replaces `break` (or `continue`) by the function break_out_to_level_n(j), for $2 \leq j \leq n$. When a **SBRn** mutant executes, the execution of the mutated statement causes the loop, inside which the mutated statement is nested, and the j enclosing loops, to terminate.

Let S' denote the loop immediately enclosing a `break` or a `continue` statement and nested n, $n > 0$, levels inside the loop S in function f. The **SBRn** operator encourages the tester to construct a test case t to show that $E^s(S, t)$ is a DES of f with respect to $P_f(t)$ and the mutated S. The exact expression for $E(S, t)$ can be derived for f and its mutant from the execution sequence construction rules listed in Section 8.10.3.

The **SBRn** operator has no effect on the following constructs:

- `break` or `continue` statements that are nested only one level deep.
- A `break` intended to terminate the execution of a `switch` statement. Note that a `break` inside a loop nested in one of the cases of a `switch`, is subject to mutation by **SBRn** and **SBRC**.

Continue Out to nth Enclosing Level

This operator is similar to **SBRn**. It replaces a nested `break` or a `continue` by the function continue_out_to_level_n(j), $2 \leq j \leq n$.

The **SCRn** operator has no effect on the following constructs:

- `break` or `continue` statements that are nested only one level deep.
- A `continue` statement intended to terminate the execution of a `switch` statement. Note that a `continue` inside a loop nested in one of the cases of a `switch` is subject to mutation by **SCRn** and **SCRB**.

`while` *Replacement by* `do-while`

Though a rare occurrence, it is possible that a `while` is used instead of a `do-while`. The **SWDD** operator models this fault. The `while` statement is replaced by the `do-while` statement.

Example 8.34 Consider the following loop:

/* This loop is from p 69 of Kernighan and Ritchie's book. */

```
while (– –lim>0 && (c=getchar()) != EOF && c !='\n')
  [i++]=c;                                                    F5
```

When the **SWDD** operator is applied, the above loop is mutated to the following.

```
do {
  s[i++]=c;                                                   M14
}
  while (– –lim>0 && (c=getchar()) != EOF && c !='\n')
```

`do-while` *Replacement by* `while`

The `do-while` statement may have been used in a program when the `while` statement would have been the correct choice. The **SDWD** operator models this fault. A `do-while` statement is replaced by a `while` statement.

Example 8.35 Consider the following `do-while` statement in P:

/* This loop is from p 64 of Kernighan and Ritchie's book.*/

```
do {
  s [ i++ ] = n % 10 + '0';                                   F6
}while (( n /= 10) > 0);
```

It is mutated by the **SDWD** operator to the following loop.

```
  while (( n /= 10) > 0) {
    s [ i++ ] = n % 10 + '0';                                 M15
}
```

Notice that the only test data that can distinguish the above mutant is one that sets n to 0 immediately prior to the loop execution. This test case ensures that $E(S, *)$, S being the original `do-while`

statement, is a DES for $P_S(t)$ with respect to the mutated statement, i.e. the `while` statement.

Multiple Trip Trap

For every loop in P, we would like to ensure that the loop body satisfied the following two conditions:

- C1: the loop has been executed more than once.
- C2: the loop has an effect on the output of P.

The **STRP** operator replaces the loop body with the trap_on_statement. A test case that distinguishes such a mutant implies that the loop body has been executed at least once. However, this does not ensure two conditions mentioned above. The **SMTT** and **SMTC** operators are designed to ensure C1 and C2.

The **SMTT** operator introduces a guard in front of the loop body. The guard is a logical function named trap_after_n^{th}_loop_iteration(n). When the guard is evaluated the nth time through the loop, it distinguishes the mutant. The value of n is decided by the tester.

Example 8.36 Consider the following `for` statement:

/* This loop is taken from p 87 of Kernighan and Ritchie's book. */

```
for (i = left+1; i <= right; i++)
 if (v [ i ] < v [ left ])                                  F7
  swap (v, ++ last, i);
```

Assuming that $n = 2$, this will be mutated by the **SMTT** operator to the following.

```
for (i = left+1; i ≤ right; i++)
 if (trap_after_nth_loop_iteration)(2){
  if (v [ i ] < v [left ])                                  M16
   swap (v, ++ last, i);
 }
```

For each loop in the program under test, the **SMTT** operator encourages the tester to construct a test case so that the loop is iterated at least twice.

Multiple Trip Continue

An **SMTT** mutant may be distinguished by a test case that forces the mutated loop to be executed twice. However, it does not ensure condition C2 mentioned earlier. The **SMTC** operator is designed to ensure C2.

SMTC introduces a guard in front of the loop body. The guard is a logical function named false_after_n^{th}_loop_iteration(n). During the first n iterations of the loop, false_after_n^{th}_loop_iteration() evaluates to *true*, thus letting the loop body execute. During the $(n+1)^{th}$ and subsequent iterations, if any, it evaluates to *false*. Thus a loop mutated by **SMTC** will iterate as many times as the loop condition demands. However, the loop body will *not* be executed during the second and any subsequent iteration.

Example 8.37 The loop in F7 is mutated by the **SMTC** operator to the loop in M17.

```
for (i = left+1; i ≤ right; i++)
 if (false_after_nth_loop_iteration()){
  if (v [ i ] < v [left ])                                  M17
   swap (v, ++ last, i);
 }
```

The **SMTC** operator may generate mutants containing infinite loops. This is specially true when the execution of the loop body effects one or more variables used in the loop condition.

For a function f, and each loop S in RAP(f), **SMTC** encourages the tester to construct a test case t which causes the loop to be executed more than once such that $E(f,t)$ is a DES of $P_f(t)$ with respect to the mutated loop. Note that **SMTC** is stronger than **SMTT**. This implies that a test case that distinguishes an **SMTC** mutant for statement S, will also distinguish an **SMTT** mutant of S.

Sequence Operator Mutation

Use of the *comma* operator results in the left to right evaluation of a sequence of expressions and forces the value of the rightmost expression to be the *result*. For example, in the statement f (a, (b = 1, b + 2), c), function f has three parameters. The second parameter has the value 3. The programmer may use an incorrect sequence of expressions thereby forcing the incorrect value to be the result. The **SSOM** operator is designed to model this fault.

Let $e_1, e_2, \ldots, e_n$ denote an expression consisting of a sequence of n sub-expressions. According to the syntax of C, each e_i can be an

assignment-expression separated by the *comma* operator. The **SSOM** operator generates $(n - 1)$ mutants of this expression by rotating left the sequence one sub-expression at a time.

Example 8.38 Consider the following statement:

```
/* This loop is taken from p 63 of Keringuan and Ritchie's book. */

for (i = 0, j = strlen(s) – 1; i < j; i++, j– –)                    F8
  c = s[i], s[i] = s[j], s[j] = c;
```

The following two mutants are generated when the **SSOM** operator is applied on the body of the above loop.

```
for (i = 0, j = strlen(s) –1; i < j; i++, j– –)
/* One left rotation generates this mutant. */                       M18
  s[i] = s[j], s[j] = c, c = s[i];
for (i =0, j = strlen(s) –1; i < j; i++, j– –)

/* Another left rotation generates this mutant. */                   M19
  s[j] = c, c = s[i], s[i] = s[j];
```

When **SSOM** is applied to the `for` statement in the above program, it generates two additional mutants, one by mutating the expression (i = 0, j = strlen(s) – 1) to (j = strlen (s) – 1 , i = 0), and the other by mutating the expression (i++, j––) to (j––, i++).

The **SSOM** operator is likely to generate several mutants equivalent to the parent. The mutants generated by mutating the expressions in the `for` statement in the above example, are equivalent. In general, if the subexpressions do not depend on each other then the mutants generated will be equivalent to their parent.

Move Brace Up or Down

The closing brace (}) is used in C to indicate the end of a compound statement. It is possible for a programmer to incorrectly place the closing brace thereby including, or excluding, some statements within a compound statement. The **SMVB** operator models this fault.

A compound statement might incorrectly include or exclude another statement or a statement sequence. This error is modeled by the SMVB operator.

A statement immediately following the loop body is pushed inside the body. This corresponds to moving the closing brace *down* by one statement. The last statement inside the loop body is pushed out of the body. This corresponds to moving the closing brace *up* by one statement.

The semicolon is considered as the ending brace in a compound statement without opening and closing braces.

A compound statement that consists of only one statement may not have explicit braces surrounding it. However, the beginning of a compound statement is considered to have an implied opening brace and the semicolon at its end is considered to be an implied closing brace. To be precise, the semicolon at the end of the statement inside the loop body is considered as a semicolon *followed by a closing brace*.

Example 8.39 Consider again the function *trim* from Kernighan and Ritchie's book.

/* This function is from Kernighan and Ritchie's book. */

```
 int trim(char s[ ] )
S1 {
     int n;                                                    F9
S2   for (n = strlen(s)−1; n >= 0; n− −)
S3    if (s[n] != ' '&& s[n] != '\t' && s[n] != ' \n')
S4     break;
S5    s[n+1] = '\0';
S6     return n;
    }
```

The following two mutants are generated when *trim* is mutated using the **SMVB** operator.

/* This is a mutant generated by **SMVB**. In this one, the `for` loop body extends to include the s [n+1] = '\0' statement. */

```
int trim(char s[ ] )
{
 int n;
 for (n = strlen(s)−1; n >= 0; n− −) {                         M20
  if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
   break;
 s[n+1] = '\0';
 }
 return n;
}
```

/* This is another mutant generated by **SMVB**. In this one the `for` loop body becomes empty. */

```
int trim(char s[ ] )
{
```

```
  int n;
  for (n = strlen(s)−1; n >= 0; n− −);                    M21
  if (s[n] != ' ' && s[n] != '\t' && s[n]!= '\n')
   break;
  s[n+1] = '\0';
  return n;
}
```

In certain cases, moving the brace may include, or exclude, a large piece of code. For example, suppose that a `while` loop with a substantial amount of code in its body, follows the closing brace. Moving the brace down will cause the entire `while` loop to be moved into the loop body that is being mutated. A C programmer is not likely to make such an error. However, there is a good chance of such a mutant being distinguished quickly during mutant execution.

Switch Statement Mutation

Errors in the formulation of the cases in a `switch` statement are modeled by **SSWM**. The expression e in the `switch` statement is replaced by the trap_on_case function. The input to this function is a condition formulated as $e = a$, where a is one of the case labels in the `switch` body. This generates a total of n mutants of a `switch` statement assuming that there are n case labels. In addition, one mutant is generated with the input condition for trap_on_case set to $e = d$, where d is computed as $d = e! = c_1 \&\& e! = c_2 \&\& \ldots e! = c_n$. The next example exhibits some mutants generated by **SSWM**.

Errors in the construction of the switch statement are modeled using the SSWM operator.

Example 8.40 Consider the following program fragment:

/* This fragment is from a program of Kernighan and Ritchie's book.

```
switch(c) {
case '0': case '1': case '2': case '3':case '4': case '5':
case '6':case '7':case '8':case '9':
 ndigit[c-'0']++;
 break;
case ' ':
case '\n':
case '\t':                                                F10
 nwhite++;
 break;
default:
```

```
  nother++;
  break;
}
```

The **SSWM** operator will generate a total of 14 mutants for F10. Two of them appear in M22 and M23.

```
switch(trap_on_case(c,'0'))  {
case '0': case '1': case '2': case '3':case '4': case '5':
case '6':case '7':case '8':case '9':
  ndigit[c−'0']++;
  break;
case ' ':
case '\n':
case '\t':                                                    M22
  nwhite++;
  break;
default:
  nother++;
  break
}
```

c'=c; /* This is to ensure that side effects in c occur once. */

```
d = c'!= '0' && c'!= '1' && c'!= '3' && c'!= '4'
  && c'!= '5' &&
  c'!= '6' && c'!= '7' && c'!= '8' &&
  c'!= '9' && c'!= '\n' && c'!= '\t';
switch(trap_on_case(c', d))  {
⋮
/*   switch  body  is the same as that in M22.  */              M23
⋮
}
```

A test set that distinguishes all mutants generated by **SSWM** ensures that all cases, including the default case, have been covered. We refer to this coverage as *case coverage*. Note that the **STRP** operator may not provide case coverage especially when there is fall-through code in the `switch` body. This also implies that some of the mutants generated when **STRP** mutates the cases in a `switch` body, may be equivalent to those generated by **SSWM**.

Example 8.41 Consider the following program fragment:

/* This is an example of fall-through code. */

```
switch (c) {
case '\n':
 if (n==1) {
  n--;
  break;                                    F11
 }
 putc('\n');
case '\r':
 putc('\r');
 break;
}
```

One of the mutants generated by **STRP** when applied on F11 will have the putc('\r') in the second case replaced by trap_on_statement(). A test case that forces the expression c to evaluate to '\n' and n evaluate to any value not equal to 1, is sufficient to kill such a mutant. On the contrary, an **SSWM** mutant will encourage the tester to construct a test case that forces the value of c to be '\r'.

It may, however, be noted that both the **STRP** and the **SSWM** serve different purposes when applied on the `switch` statement. Whereas **SSWM** mutants are designed to provide case coverage, mutants generated when **STRP** is applied to a `switch` statement, are designed to provide statement coverage *within* the `switch` body.

8.10.11 Mutating program variables

Incorrect use of identifiers can often induce program faults that remain unnoticed for quite long. Variable mutations are designed to model such faults. Table 8.7 lists all variable mutation operators for C.

Operators that mutate program variables model errors in the use of variable names. Such operators often lead to a large number of mutants.

Scalar Variable Reference Replacement

Use of an incorrect scalar variable is the fault modeled by the two mutant operators: **VGSR** and **VLSR**. **VGSR** mutates all scalar variable references by using GS'_f as the range. **VLSR** mutates all scalar variable references by using LS'_f as the range of the mutation operator. Types are ignored during scalar variable replacement. For example, if i is an integer and x a real, i will be replaced by x and vice versa.

Entire scalar references are mutated. For example, if *screen* is as declared in Example 8.26, and $screen.p1.x$ is a reference, then the entire reference, i.e. $screen.p1.x$, will be mutated. $p1$ or x will not be mutated separately by any one of these two operators. The individual components of a structure

Table 8.7 List of mutation operators for variables in C.

Mutop	Domain	Description
VASM	Array subscript	Array reference subscript mutation
VDTR	Scalar reference	Absolute value mutation
VGAR	Array reference	Mutate array references using global array references
VGLA	Array reference	Mutate array references using both global and local array references
VGPR	Pointer reference	Mutate pointer references using global pointer references
VGSR	Scalar reference	Mutate scalar references using global scalar references
VGTR	Structure reference	Mutate structure references using global structure references
VLAR	Array reference	Mutate array references using local array references
VLPR	Pointer reference	Mutate pointer references using local pointer references
VLSR	Scalar reference	Mutate scalar references using local scalar references
VLTR	Structure reference	Mutate structure references using local structure references
VSCR	Structure component	Structure component replacement
VTWD	Scalar expression	Twiddle mutations

may be mutated by the **VSCR** operator. *screen* itself may be mutated by one of the strucuture reference replacement operators. We often say that an entity x may be mutated by an operator. This implies that there may be no other entity y to which x can be mutated. Similarly, in a reference such as $*p$, for p as declared above, $*p$ will be mutated. p alone may be mutated by one of the pointer reference replacement operators. As another example, the entire reference $q[i]$ will be mutated, q itself may be mutated by one of the array reference replacement operators.

Array Reference Replacement

Incorrect use of an array variable is the fault modeled by two mutation operators: **VGAR** and **VLAR**. These operators mutate an array reference

in function f using, respectively, the sets GA'_f and LA'_f. Types are preserved while mutating array references. Here, name equivalence of types as defined in C is assumed. Thus, if a and b are, respectively, arrays of integers and pointers to integers, a will not be replaced by b and viceversa.

8.10.12 Structure Reference Replacement

Incorrect use of structure variables is modeled by two mutation operators **VGTR** and **VLTR**. These operators mutate a structure reference in function f using, respectively, the sets GT'_f and LT'_f. Types are preserved while mutating structures. For example, if s and t denote two structures of different types then s will not be replaced by t and viceversa. Again, *name* equivalence is used for types as in C.

Pointer Reference Replacement

Incorrect use of a pointer variable is modeled by two mutation operators **VGPR** and **VLPR**. These operators mutate a pointer reference in function f using, respectively, the sets GP'_f and LP'_f. Types are preserved while performing mutation. For example, if p and q are pointers to an integer and structure, respectively, then p will not be replaced by q, and viceversa.

Structure Component Replacement

Often one may use the wrong component of a structure. **VSCR** models such faults. Here, *structure* refers to data elements declared using the `struct` type specifier. Let s be a variable of some structure type. Let $s.c_1.c_2 \ldots .c_n$ be a reference to one of its components declared at *level n* within the structure. c_i, $1 \leq i \leq n$, denotes an identifier declared at level i within s. **VSCR** systematically mutates each identifier at level i by all the other type compatible identifiers at the same level.

Example 8.42 Consider the following structure declaration:

```
struct example {
 int x;
 int y;
 char c;                                    F12
 int d[10];
}
struct example s, r;
```

Reference $s.x$ will be mutated to $s.y$ and $s.c$ by **VSCR**. Another reference $s.d[j]$ will be mutated to $s.x$, $s.y$, and $s.c$. Note that the

reference to *s* itself will be mutated to *r* by one of **VGSR** or **VLSR** operators.

Next, suppose that we have a pointer to *example* declared as `struct` example *p; A reference such as $p->x$ will be mutated to $p->y$ and $p->c$. Now, consider the following recursive structure:

```
struct tnode {
 char *word;
 int count;
 struct tnode *left;
 struct tnode *right;                                    F13
}
struct tnode *q;
```

A reference such as $q->left$ will be mutated to $q->right$. Note that *left*, or any field of a structure, will *not* be mutated by **VGSR** or by VLSR operators. This is because a field of a structure does not belong to any of the global or local sets, or reference sets, defined earlier. Also, a reference such as $q->count$ will not be mutated by **VSCR** because there is no other compatible field in F13.

Array reference subscript mutation

While referencing an element of a multidimensional array, the order of the subscripts may be incorrectly specified. **VASM** models this fault. Let a denote an n-dimensional array, $n > 1$. A reference such as $a[e_1][e_2]\ldots[e_n]$ with e_i, $1 \le i \le n$, denoting a subscript expression, will be mutated by rotating the subscript list. Thus, the above reference generates the following $(n-1)$ mutants when **VASM** is applied:

$$
\begin{aligned}
a[e_n]\,[e_1]\ldots[e_{n-2}]\,[e_{n-1}]\\
a[e_{n-1}]\,[e_n]\ldots[e_{n-3}]\,[e_{n-2}]\\
\vdots\\
a[e_2]\,[e_3]\ldots[e_n]\,[e_1]
\end{aligned}
$$

Domain traps

The **VDTR** operator provides domain coverage for scalar variables. The domain partition consists of three subdomains: one containing negative values, one containing a zero, and one containing only the positive values.

VDTR mutates each scalar reference x of type t in an expression, by $f(x)$, where f could be one of the several functions shown in Table 8.8. Note that

Table 8.8 Functions used by the **VDTR** operator.

Function † **introduced**	**Description**
trap_on_negative_x	Mutant distinguished if argument is negative, else return argument value.
trap_on_positive_x	Mutant distinguished if argument is positive, else return argument value.
trap_on_zero_x	Mutant distinguished if argument is zero, else return argument value.

† x can be integer, real, or double. It is integer if the argument type is `int`, `short`, `signed`, or `char`. It is real if the argument type is `float`. It is double if the argument is of type `double` or `long`.

all functions listed in Table 8.8 for a type t are applied on x. When any of these functions is executed, the mutant is distinguished. Thus, if i, j, and k are pointers to integers, then the statement $*i = *j + *k++$ is mutated by **VDTR** to the following statements.

$$*i = \text{trap_on_zero_integer}(*j) + *k++$$
$$*i = \text{trap_on_positive_integer}(*j) + *k++$$
$$*i = \text{trap_on_negative_integer}(*j) + *k++$$
$$*i = *j + \text{trap_on_zero_integer}(*k++)$$
$$*i = *j + \text{trap_on_positive_integer}(*k++)$$
$$*i = *j + \text{trap_on_negative_integer}(*k++)$$
$$*i = \text{trap_on_zero_integer}(*j + *k++)$$
$$*i = \text{trap_on_positive_integer}(*j + *k++)$$
$$*i = \text{trap_on_negative_integer}(*j + *k++)$$

In the above example, $*k++$ is a reference to a scalar, therefore the trap function has been applied to the entire reference. Instead, if the reference was (*k)++, then the mutant would be $f(*k)++$, f being any of the relevant functions.

Twiddle mutations

Values of variables or expressions can often be off the desired value by ± 1. The twiddle mutations model such faults. Twiddle mutations are useful for checking boundary conditions for scalar variables.

Off-by 1 errors are modeled by the twiddle mutations.

Each scalar reference x is replaced by $pred(x)$ and $succ(x)$, where *pred* and *succ* return, respectively, the immediate predecessor and the immediate successor of the current value of the argument. When applied to a float argument, a small value is added (by *succ*) to, or subtracted (by *pred*) from, the argument. This value can be user defined, such as $\pm.01$, or may default to an implementation defined value.

Example 8.43 Consider the assignment: $p = a + b$ Assuming that p, a, and b are integers, **VTWD** will generate the following two mutants.

$$p = a + b + 1$$
$$p = a + b - 1$$

Pointer variables are not mutated. However, a scalar reference constructed using a pointer is mutated as defined above. For example, if p is a pointer to an integer, then $*p$ is mutated. Some mutants may cause overflow or underflow faults implying that they are distinguished.

8.11 Mutation Operators for Java

Java, like many others, is an object-oriented programming language. Any such language provides syntactic constructs to encapsulate data and procedures into objects. Classes serve as templates for objects. Procedures within a class are commonly referred to as *methods*. A method is written using traditional programming constructs such as assignments, conditionals, and loops.

The existence of classes, and the inheritance mechanism in Java, offers a programmer ample opportunities to make mistakes that end up in program faults. Thus the mutation operators for mutating Java programs are divided into two generic categories: traditional mutation operators and class mutation operators.

Mutation operators for Java consists of a set of traditional mutation operators found in procedural programming languages and a set of new mutation operators specific to the object-orientation of Java.

Mutation operators for Java have evolved over a period and are a result of research contributions of several people. The specific set of operators discussed here was proposed by Yu-Seung Ma, Tong-rae Kwon, and Jeff Offutt. We have decided to describe the operators proposed by this group as these operators have been implemented in the μJava (also known as muJava system) mutation system discussed briefly in Section 8.15. Sebastian Danicic from Goldsmiths College University of London has also implemented a set of mutation operators in a tool named Lava. Mutation operators in Lava belong to the "traditional mutation operators" category

described in Section 8.11.1. Yet another tool named Jester, developed by Ivan Moore, has another set of operators that also fall into the "traditional mutation operators" category.

Though a large class of mutation operators developed procedural languages such as Fortran and C are applicable to Java methods, a few of these have been selected and grouped into the "traditional" category. Mutation operators specific to the object-oriented paradigm and Java syntax are grouped into the "class mutation operators" category.

The five mutation operators in the "traditional" category are listed in Table 8.9. The class related mutation operators are further subdivided into inheritance, polymorphism and dynamic binding, method overloading, OO, and Java-specific categories. Operators in these four categories are listed in Tables 8.9 through 8.13. We will now describe the operators in each class with examples. We use the notation

$$P \underset{\Rightarrow}{\overset{\text{MutOp}}{}} Q$$

to indicate that mutant operator MutOp when applied to program segment P creates mutant Q as one of the several possible mutants.

Only a small set of traditional mutation operators have been selected for Java. This selection is based on empirical studies to determine which mutation operators are the most effective in detecting errors in programs.

8.11.1 Traditional mutation operators

Arithmetic expressions have the potential of producing values that are not expected by the code that follows. For example, an assignment $x = y + z$ might generate a negative value for x, which causes a portion of the

Table 8.9 Mutation operators for Java that model faults due to common programming mistakes made in procedural programming.

Mutop	Domain	Description
ABS	Arithmetic expressions	Replaces an arithmetic expression *e* by *abs*(*e*).
AOR	Binary arithmetic operator	Replaces a binary arithmetic operator by another binary arithmetic operator valid in the context.
LCR	Logical connectors	Replaces a logical connector by another.
ROR	Relational operator	Replaces a relational operator by another.
UOI	Arithmetic or logical expressions	Insert a unary operator such as the unary minus or the logical not.

following code to fail. The ABS operator generates mutants by replacing each component in an arithmetic expression by its absolute value. Mutants generated for $x = y + z$ are are follows:

$$x = abs(y) + z$$
$$x = y + abs(z)$$
$$x = abs(y + z)$$

The remaining traditional operators for Java have their correspondence with the mutation operators for C. The AOR operator is similar to OAAN, LCR to OBBN, ROR to ORRN, and UOI to OLNG and VTWD. Note that two similar operators, when applied to programs in different languages, will likely generate a different number of mutants.

8.11.2 Inheritence

Subclassing can be used in Java to hide methods and variables. A subclass C can declare variable x, hiding its declaration, if any, in $parent(C)$. Similarly, C can declare a method m hiding its declaration, if any, in $parent(C)$. The IHD, IHI, IOD, and IOP operators model faults that creep into a Java program due to mistakes in redeclaration of class variables and methods in a corresponding subclass.

Example 8.44 As shown next, the IHD mutation operator removes the declaration of a variable that overrides its declaration in the parent class.

```
class planet{                                  class planet{
  double dist;                                   double dist;
  ⋮                                              ⋮
}                                              }
                                   IHD
class farPlanet extends planet{    ⟹          class farPlanet extends
                                               planet{
  double dist;                                   // declaration removed.
  ⋮                                              ⋮
}                                              }
```

Removal of the declaration for `dist` in the subclass exposes the instance of `dist` in the parent class. The IHI operator does the reverse of this process by adding a declaration as in the following:

Table 8.10 Mutation operators for Java that model faults related to inheritance.

Mutop	Domain	Description
IHD	Variables	Removes the declaration of variable x in subclass C if x is declared *parent*(C).
IHI	Subclasses	Adds a declaration for variable x to a subclass C if x is declared in the *parrent*(C).
IOD	Methods	Removes declaration of method m from a subclass C if m is declared in *parent*(C).
IOP	Methods	Inside a subclass, move a call *super*.M(..) inside method m to the beginning of m, end of m, one statement down in m, and one statement up in m.
IOR	Methods	Given that method f_1 calls method f_2 in *parent*(C), rename f_2 to f_2' if f_2 is overridden by subclass C.
ISK	Accesses to parent class	Replace explicit reference to variable x in *parent*(C) given that x is overridden in subclass C.
IPC	`super` calls	Delete the `super` keyword from the constructor in a subclass C.

```
class planet{                                  class planet{
  String name;                                   String name;
  double dist;                                   double dist;
  ⋮                                              ⋮
}                                              }
                                   IHI
class farPlant extends planet{   ⟹   class farPlant extends
                                       planet{
  ⋮                                      double dist
                                         ⋮
}
```

As illustrated next, IOD exposes a method otherwise hidden in a subclass through overriding.

```
class planet{                  class planet{
  String name;                   String name;
  Orbit orbit (...);             Orbit orbit (...);
```

```
  ⋮                                          ⋮
}                                          }
                                IOD
class farPlanet extends planet{   ⟹   class farPlanet extends
                                          planet{
  Orbit orbit(...);                          // Method orbit removed.
  ⋮                                          ⋮
}                                          }
```

As shown below, the IOP operator changes the position of any call to an overridden method made using the `super` keyword. Only one mutant is shown whereas up to three additional mutants could be created by moving the call to the parent's version of `orbit` one statement up, one statement down, at the beginning, and the end of the calling method (see Exercise 8.18).

```
class planet{                              class planet{
  String name;                               String name;
  Orbit orbit (...){                         Orbit orbit (...){
  oType=high;                                oType=high;
  ⋮                                          ⋮
}                                          }
  ⋮                                          ⋮
                                IOP
}                               ⟹       }
class farPlant extends planet{             class farPlant extends planet{
  Orbit orbit (...);                         Orbit orbit (...);
  oType=low; super.orbit( )                  ⋮
  ⋮                                          super.orbit( ); oType=low;
}                                          ⋮
                                           }
```

As shown next, method `orbit` in class `planet` calls method `check`. The IOR operator renames this call to, say, `j_check`.

```
class planet{                          class planet{
  String name;                           String name;
  Orbit orbit (...)                      Orbit orbit (...)
  {... check( );...};                    {... j_check( );...};
  void check (...){...}                  void j_check (...)
  ⋮                          IOR         ⋮
}                            ⟹        }
```

```
class farPlanet extends planet{
  void check (...){...}
  ⋮
}
```

```
class farPlanet extends
planet{
  void check (...){...}
  ⋮
}
```

Given an object `X` of type `farPlanet`, `X.orbit( )` will now invoke the subclass version of `check` and not that of the parent class. Distinguishing an IOR mutant requires a test case to show that `X.orbit( )` must invoke `check` of the parent class and not that of the subclass.

The ISK operator generates a mutant by deleting, one at a time, occurrences of the `super` keyword from a subclass. This forces the construction of a test case to show that indeed the parent's version of the hidden variable or method is intended in the subclass and not the version declared in the subclass.

```
class farPlanet extends planet{
  ...
  p=super.name
  ⋮
}
```

ISK ⟹

```
class farPlanet extends
planet{
  ...
  p=name
  ⋮
}
```

Lastly, the IPC operator in the inheritance fault modeling category creates mutants by replacing any call to the default constructor of the parent class.

```
class farPlant extends planet{
  ...
  farPlanet (String p);
  ...
  super(p)
  ⋮
}
```

IPC ⟹

```
class farPlant extends
planet{
  ...
  farPlanet (String p);
  ...
  // Call removed.
  ⋮
}
```

This completes our illustration of all mutation operators that model faults related to the use of inheritance mechanism in Java. Exercise 8.17 asks you to develop conditions that a test case must satisfy to distinguish mutants created by each operator.

8.11.3 Polymorphism and dynamic binding

Mutation operators related to polymorphism model faults related to Java's polymorphism features. The operators force the construction of test cases

Table 8.11 Mutation operators for Java that model faults related to polymorphism and dynamic binding.

Mutop	Domain	Description
PMC	Object instantiations	Replace type t_1 by t_2 in object instantiation via `new`; t_1 is the parent type and t_2 the type of its subclass.
PMD	Object declarations	Replace type t_1 of object x by type t_2 of its parent.
PPD	Parameters	For each parameter, replace type t_1 of parameter object x by type t_2 of its parent.
PRV	Object references	Replace the object reference, e.g. O_1, on the right side of an assignment by a reference to a type compatible object reference O_2, where both O_1 and O_2 are declared in the same context.

that ensure that the type bindings used in the program under test are correct and the other syntactically possible bindings are either incorrect or equivalent to the ones used.

Table 8.11 lists the four mutation operators that model faults due to incorrect bindings. The following example illustrates these operators.

Example 8.45 Let `planet` be the parent of class `farPlanet`. The PMC and PMD mutation operators model the incorrect use of a parent or its subclass as object types. The PPD operator models an incorrect parameter-type binding with respect to a parent and its subclass. Here are sample mutants generated by these two operators.

```
 farPlanet p;                PMC   planet p;
   p=new farPlanet( )   ⟹          p=new farPlanet( )
 planet p;                 PMD   planet p;
   p=new planet( )      ⟹          p=new farPlanet( )
void launchOrbit                 void launchOrbit
    (farPlanet p){                   (planet p){
...                        PPD   ...
};                         ⟹    };
```

The PRV operator models a fault in the use of an incorrect object but of a compatible type. The next example illustrates a mutant

generated by replacing an object reference on the right side of an assignment by another reference to an object that belongs to a subclass of the object on the left. Here we assume that `Element` is the parent class of `specialElement` and `gas`.

```
Element anElement;                    Element anElement;
specialElement sElement;   PRV        specialElement sElement;
gas g;                     ⟹         gas g;
...                                   ...
anElement=sElement;                   anElement=g;
```

8.11.4 Method overloading

Method overloading allows a programmer to define two or more methods within the same class but with different signatures. The signatures allow a compiler to distinguish between the two methods. Thus a call to an overloaded method *m* is translated to the appropriate method based on signature matching.

Overloading also offers a programmer an opportunity to make a mistake and use an incorrect method. A compiler will not be able to detect the fault due to such a mistake. The method overloading operators, listed in Table 8.12, are designed to model such faults.

Example 8.46 A mutant generated by the OMR operator forces a tester to generate a test case that exhibits some difference in the behavior of two or more overloaded methods. This is accomplished by replacing the body of each overloaded method with that of another. Here is a sample overloaded method and the two OMR mutants.

Table 8.12 Mutation operators for Java that model faults related to method overloading.

Mutop	Domain	Description
OMR	Overloaded methods	Replace body of one overloaded method by that of another.
OMD	Overloaded methods	Delete overloaded method.
OAO	Methods	Change the order of method parameters.
OAN	Methods	Delete arguments in overloaded methods.

```
void init (int i){...};                 OMR   void init (int i){...};
void init (int i, String s){...};       ⟹    void init (int i,
                                              String s){
                                                this.init (i);
                                              }
void init (int i){...};                       void init (int i){
void init (int i, String s){...};       OMR     this.init (i);
                                        ⟹    }
                                              void init (int i,
                                                  String s){...};
```

The OMD operator generates mutants by deleting overloaded methods one at a time. Test adequate with respect to OMD mutants ensures complete coverage of all overloaded methods. Note that some of the OMD mutants might fail to compile (Can you see why ?). Here is a sample program and one of its two OMD mutants.

```
void init (int i){...};                 OMD   void init (int i){...};
void init (int i, String s){...};       ⟹    // Second init deleted. }
```

Two overloaded methods might have the same number of parameters but possess a different signature. This raises the potential for a fault in the use of an overloaded method. The OAO operator models this fault and generates mutants by changing the parameter order given that the mutant so created is syntactically valid.

```
Orbit.getOrbit(p, 4);   OMR ⟹   Orbit.getOrbit(4, p);
```

Note that in case of three parameters, more than one OAO mutant might be generated. An overloaded method with fewer (or more) parameters might be used instead of a larger (or fewer) number of parameters. This fault is modeled by the OAN operator.

```
Orbit.getOrbit(p, 4);   OMR ⟹   Orbit.getOrbit(p);
```

Again, note that the generated mutant must be syntactically valid. Furthermore, one could think of at least two mutants of the call to the `getOrbit` method, only one of which is shown here.

8.11.5 Java specific mutation operators

Some mutation operators are specific to Java. These include the JTD that deletes the "this" keyword and a few others.

A few additional operators model Java language specific faults and common programming mistakes. Eight such operators are listed in Table 8.13.

The JTD operator deletes `this` keyword. This forces a tester to show that indeed the variable or method referenced via `this` was intended. The JSC operator removes or adds the `static` keyword to model faults related to the use of instance verses class variables. The JID operator removes any

initializations of class variables to model variable initialization faults. The JDC operator removes programmer implemented constructors. This forces a tester to generate a test case that demonstrates the correctness of the programmer supplied constructors.

Four operators model common programming mistakes. Using an object reference instead of the content of an object is a fault modeled by the EOA operator.

Example 8.47 The EOA operator models the mistake illustrated next.

```
Element hydrogen,                    Element hydrogen, hisotope;
  hisotope;
hydrogen=new Element ();   EOA
                           ⟹        hydrogen=new Element ();
hisotope=hydrogen;                   hisotope=hydrogenclone( );
```

There exists potential for a programmer to invoke an incorrect accessor or a modifier method. These faults are modeled by the EAM and EMM operators, respectively. The accessor (modifier) methods names are replaced by some other existing name of an accessor (modifier) method that matches its signature.

Table 8.13 Mutation operators for Java that model language specific and common faults.

Mutop	Domain	Description
JTD	`this`	Delete `this` keyword.
JSC	Class variables	Change a class variable to an instance variable.
JID	Member variables	Remove the initialization of a member variable.
JDC	Constructors	Remove user-defined constructors.
EOA	Object references	Replace reference to an object by its contents using *clone*( ).
EOC	Comparison expressions	Replace == by *equals*.
EAM	Calls to accessor methods	Replace call to an accessor method by a call to another compatible accessor method.
EMM	Calls to modifier methods	Replace call to a modifier method by a call to another compatible modifier method.

Example 8.48 The following two examples illustrate the EAM and EMM operators.

```
hydrogen.getSymbol();  ⟹(EAM)  hydrogen.getAtNumber();
hydrogen.setSymbol();  ⟹(EMM)  hydrogen.setAtNumber();
```

8.12 Comparison of Mutation Operators

Mutation operator set for C is by far the largest. However, that for Java was designed taking into account the effectiveness of mutation operators and hence is relatively small.

C has a total of 77 mutation operators compared to 22 for Fortran 77, hereafter referred to as Fortran, and 29 for Java. Note that the double appearance of the number 77 is purely coincidental. Table 8.14 lists all the Fortran mutation operators and the corresponding semantically nearest C and Java operators.

Fortran is one of the earliest languages for which mutation operators were designed by inventors of program mutation and their graduate students. Hence the set of 22 operators for Fortran is often referred to as "traditional mutation operators."

Recall from Section 8.5.2 that there is no perfect, or optimal, set of mutation operators for any language. Design of mutation operators is an art as well as a science. Thus one could always justifiably come up with a mutation operator not presented in this chapter. With this note of freedom, let us now take a comparative look at the sets of mutation operators designed and implemented for Fortran, C, and Java, and understand their similarities and differences.

In an empirical study the ABS and ROR operators were found most effective in the detection of faults.

1. Empirical studies have shown that often a small set of mutation operators is sufficient to obtain a strong test set capable of detecting a large number of program faults. For example, in one study, mutants generated by applying only the ABS and ROR operators to five programs were able to detect 87.67% of all faults.

   Similar data from several empirical studies has led to the belief that mutation testing can be performed efficiently and effectively, using only a small set of operators. This belief is reflected in the small set of five traditional mutation operators for Java. While each one of the operators listed in Table 8.14 can be used to mutate Java programs, several have not been included in the set listed in Section 8.11.
2. C has many more primitive types than Fortran. Further, types in C can be mixed in a variety of combinations. This has resulted in a large number of mutation operators in C that model faults related to an incorrect use of an operator in a C program. Similar mistakes can also be made by

Table 8.14 A comparison of mutation operators proposed for Fortran, C, and Java.

Fortran 77	Description	C	Java
AAR	Array reference for array reference	**VLSR**, **VGSR**	None
ABS	Absolute value insertion	**VDTR**	ABS
ACR	Array reference for constant replacement	**VGSR**, **VLSR**	None
AOR	Arithmetic operator replacement	**OAAN**	AOR
ASR	Array reference for scalar variable replacement	**VLSR**, **VGSR**	None
CAR	Constant for array reference replacement	**CGSR**, **CLSR**	None
CNR	Comparable array name replacement	**VLSR**, **VGSR**	None
CRP	Constant replacement	**CRCR**	None
CSR	Constant for scalar replacement	**CGSR**, **CLSR**	None
DER	DO statement *END* replacement	**OTT**	None
DSA	DATA statement alterations	None	None
GLR	GOTO label replacement	**SGLR**	None
LCR	Logical connector replacement	**OBBN**	LCR
ROR	Relational operator replacement	**ORRN**	ROR
RSR	Return statement replacement	**SRSR**	None
SAN	Statement Analysis	**STRP**	None
SAR	Scalar variable for array reference replacement	**VLSR**, **VGSR**	None
SCR	Scalar for constant replacement	**VLSR**, **VGSR**	None
SDL	Statement deletion	**SSDL**	None
SRC	Source constant replacement	**CRCR**	None
SVR	Scalar variable replacement	**VLSR**, **VGSR**	None
UOI	Unary operator insertion	**OLNG**, **VTWD**	UOI

a Java programmer, such as the replacement of the logical and operator (&&) by another version (&) that forces both operands to be evaluated. Hence, several mutation operators that mutate the operators in a C expression are also applicable to Java programs. The Lava mutation system does include an extensive set of operator mutations.

Certainly, many of the C operators that could be used to mutate Java programs are not included in Java. The OEAA operator is one such example; others can be found by browsing through the mutation operators listed in Section 8.10.

3. The statement structure of C is recursive. Fortran has *single line* statements. Though this has not resulted in more operators being defined for C, it has, however, made the definition of operators such as **SSDL** and **SRSR** significantly different from the definition of the corresponding operators in Fortran. Java does not include such statement mutations.
4. Scalar references in C can be constructed non-trivially using functions, pointers, structures, and arrays. In Fortran, only functions and arrays can be used to construct scalar references. This has resulted in operators such as **VSCR** and several others in the variable replacement category. Note that in Fortran, **SVR** is one operator whereas in C, it is a set of several operators.
5. C has a *comma* operator that is not in Fortran. This has resulted in the **SSOM** operator. This operator is not applicable to Java.
6. All iterations, or the current iteration, of a loop can be terminated in C using, respectively, the `break` and `continue` statements. Fortran provides no such facility. This has resulted in additional mutant operators such as **SBRC**, **SCRB**, and **SBRn**. Java also provides the `break` and `continue` statements and hence these operators are valid for Java.
7. The inter-class mutation operators for Java are designed to mutate programs written in object-oriented languages and hence are not applicable to Fortran and C. Of course, some of these operators are Java specific.

Many believe that mutation testing is applicable only during unit testing, However, selective use of mutation operators and code segments with a large application can help in the use of mutation for testing complex applications.

8.13 Mutation Testing within Budget

The test adequacy criteria provided by mutation has been found experimentally to be the most powerful of all the existing test adequacy criteria. This strength of mutation implies that given a test set adequate with respect to mutation, it is very likely to be adequate with respect to any other adequacy criterion as well. It also implies that if a test set has been found adequate with respect to some other adequacy criterion, then it is not adequate with respect to some mutation-based adequacy criterion. Note that

the adequacy criterion provided by mutation depends on the mutation operators used the more the operators used, the stronger is the criterion assuming that the operators are chosen carefully.

The strength of mutation does not come for free. Let C_A be the cost of developing a test set adequate with respect to criterion A. One component of C_A is the time spent by testers in obtaining an A-adequate test set for the application under test. Another component of C_A is the time spent in the generation, compilation, and execution of mutants. C_{mut}, the cost of mutation, turns out to be much greater than C_A for almost any other path oriented adequacy criterion A. We need strategies to contain C_{mut} to within our budget.

There are several strategies available to a tester to reduce C_{mut} and bring it down to an acceptable level within an organization. Some of these options are exercised by the tester while others by the tool used for assessment. Here we briefly review two strategies that have been employed by testers to reduce the cost of mutation testing. The strategy of combining mutants into one program to reduce compilation, and executing the combined program in parallel, to reduce the total execution time, have been proposed and are alluded to only in the bibliography section.

8.13.1 Prioritizing functions to be mutated

Suppose that you are testing an application P that contains a large number of classes. It is recommended that you prioritize the classes such that classes that support the most critical features of the application receive higher priority than others. If your application is written in a procedure oriented language such as C, then prioritize the functions in terms of their use in the implementation of critical features. Note that one or more classes, or one or more functions, might be involved in the implementation of a feature. Figure 8.3 illustrates class prioritization based on the criticality of features.

Functions, or classes, within a program could be prioritized for testing and hence for mutation. The prioritization may be based on code criticality and testing budget.

Now suppose that T_P is the test whose adequacy is to be assessed. We assume that P behaves in accordance with its requirements when tested against T_P. Use the mutation tool to mutate only the highest priority classes. Now apply the assessment procedure shown in Figure 8.1. At the conclusion of this procedure, you would have obtained the mutation score for T_P with respect to the mutated class. If T_P is found adequate then you could either terminate the assessment procedure or repeat the procedure by mutating the classes with the next highest priority. If T_P is not adequate then you need to construct T'_P by adding tests to T_P so that T'_P is adequate.

One can use the above strategy to contain the cost of mutation testing. At the end of the process, you would have developed an extremely reliable

Figure 8.3 Feature F_1 is accorded higher priority than feature F_2. Hence assign higher priority to classes C_1 and C_2 for mutation. Feature priority is based on criticality, i.e. on the severity of damage in case the program fails when executing a feature.

test set with respect to the critical features of the application. One major advantage of this strategy is that it allows a test manager to tailor the test process to a given budget. The higher the budget, the more classes you will be able to mutate. This strategy also suggests that mutation testing can be scaled to any sized application by a careful selection of portions of the code to be mutated. Of course, you might have guessed that a similar approach can also be used when measuring adequacy with respect to any code coverage-based criterion.

8.13.2 Selecting a subset of mutation operators

Most mutation tools offer a large palette of mutation operators to select from. In an extreme case, you could let the tool apply all operators to the application under test. This is likely to generate an overwhelmingly large set of mutants depending on the size of *P*. Instead, it is recommended that you select a small subset of mutation operators and mutate only the selected portions of *P* using these operators. An obvious question is: *What subset of mutation operators should one select?*

At least two empirical studies have shown that a small set of mutation operators can lead to a large mutation score. This indicates that most other operators do not contribute much to the mutation score.

An answer to the question above is found in Table 8.15. The data in this table is based on two sets of independently conducted experiments. One experiment was reported by Jeff Offutt, Greg Rothermel and Christian Zapf in May of 1993. The other experiment was reported in the by Eric Wong in his doctoral thesis published in December of 1993. In both studies it was found that a test set adequate with respect to a small set of mutation operators, listed in the table, is nearly adequate with respect to a much larger set of operators.

Offutt and colleagues found that by using the set of five mutation operators in the table, they could achieve an adequacy of greater than 0.99 on

Table 8.15 A constrained set of mutation operators sufficient to achieve a mutation score close to 1.

Study	Sufficient set of mutation operators
Offutt, Rothermel, and Zapf	ABS, ROR, LCR, UOI, AOR
Wong	ABS, ROR

all 22 operators listed in Table 8.14. Wong found that using the ABS and ROR operators alone, one could achieve an adequacy of over 0.97 against all 22 operators in Table 8.14. The operators are listed in Table 8.15 in the order of their effectiveness in obtaining a high overall mutation score. Similar results have also been reported earlier and are cited in the bibliography section.

The results of the two studies mentioned above suggest a simple strategy: assess the adequacy of your test set against only a small subset of all mutation operators. One might now ask: *Why do we need a larger set of operators?* As each mutation operator models a common error, it cannot be considered redundant. However, the two studies mentioned above suggest that some operators are "stronger" than the others in the sense that if a test set distinguishes all mutants generated by these then it will likely distinguish mutants generated by the "weaker" ones.

In practice, the mutation operators applied will depend on the budget. A lower budget test process will likely apply only a few operators than a higher budget process. In a lower budget situation, the operators in Table 8.15 turn out to be a useful set.

8.14 CASE and Program Testing

CASE: Computer-Aided Software Engineering is a collection of methods and tools for software development. There exists a large number of commercial and open source tools that aid in the software development processes.

Figure 8.4 shows the stages of a development process. The arrows indicate a sequence of different development stages. However, for the discussion below, the sequence does not matter and feedback loops in the process are expected. Below are some ways in which mutation can be used during the entire development process.

Requirements gathering: Software testing could begin, and does so in some processes, in the requirements analysis phase. In this phase tests

Figure 8.4 Software artifacts generated in a development process. Artifacts that can be subjected to mutation are indicated with an M on the upper right corner. Solid arrows indicate flow of the development process. Dashed arrows indicate production of artifacts in the corresponding stage.

are developed alongside system requirements. As no code is available, it is unlikely that mutation could be used at this point in the development process.

Analysis and design: The next step is requirements analysis and design. Depending on the level of formality in this process, the design could be expressed using formal specifications such as in the Z language, or as a finite state model, or as a statechart model. In all cases it is possible to apply mutation to the design and generate alternate mutated designs. Tests can be generated, automatically or manually, from the original designs. These tests can then be used to test the mutated designs to ensure the correctness of the original design.

Unit development and testing: In a traditional development environment coding could begin soon after the design has been validated via some means, e.g., design reviews. Developers and testers may test test their code at this level using tools such as JUnit, PyUnit, and other unit testing tools. At this point the adequacy of unit tests could be determined using tools such as muJava for Java and Milu for C. The adequacy score can then be used to improve unit tests. Of course as determining an accurate mutation

Some mutation testing tools, such as MuJava, have been integrated with the Eclipse development environment.

score is a time-consuming task, it should be undertaken only when the unit under test is sufficiently stable and unlikely to change too often.

Integration and testing: Units are gradually integrated into subsystems and tested as explained in Chapter 11. At this stage interface mutation could be applied to assess the adequacy of integration tests. Interface mutation focuses only on those elements of a subsystem that correspond to the component interfaces. This reduces the cost of mutation testing.

System testing: Mutation can be applied during system testing though it ought to be done carefully and in a planned manner. Let us assume that a system has been tested and has met the United States Department of Defense DO 178B requirements. For Level C (catastrophic) software these requirements ensure that the tests are MC/DC adequate.

In the above case, one could certainly attempt to apply constrained (or selective) mutation to the entire system and assess the adequacy of the system tests. However, depending on the size and complexity of the system software mutating the entire system might not be practical. In such a situation, it is best to identify portions of the system code that are most critical for reliable system operation (see Section 8.13.1). Only these portions of the code can be subject to mutation. The MC/DC adequate tests that correspond to the selected portion of the code can then be assessed and improved if found necessary.

Agile development: Testing is a nearly continuous activity in an agile development environment and depends, in at least some cases, on customer demands. Thus, customer demand on software reliability, rather than prevalent customs, is a key factor differentiating testing in an agile environment. Nevertheless, when needed mutation can be applied at all stages. However, as mentioned earlier, it should be applied only to stable code regardless of the development stage. The Jumble mutation tool has been designed to be applicable to Java code under test using JUnit in an agile environment.

8.15 Tools

A variety of mutation testing tools are available. Most are in the public domain and can be obtained by contacting their developers.

Tool support is essential while assessing the adequacy of tests for even small programs, say, 100 lines of C code. This aspect of mutation testing was known to its original developers. Consequently, research in program mutation was accompanied by the development of prototype or fully operational robust tools.

Table 8.16 A partial list of mutation tools developed and their availability.

Language	Tool	Year	Availability
Fortran	PIMS	1976	Not available
	FMS	1978	Not available
	PMS	1978	Not available
	EXPER	1978	Not available
	Mothra	1988	`http://www.isse.gmu.edu/faculty/ofut/rsrch/mut.html`
COBOL	CMS.1	1980	Not available
C	Milu	2008	`www.dcs.kcl.ac.uk/pg/jiayue/milu/`
	Proteum	1993	Available from Professor Josè `Maldonado (jcmaldon@icmc.usp.br)`
CSP	MsGAT	2001	`http://www-users.cs.york.ac.uk/~jill/ Tool.htm`
C#	Nester	2001	`http://jester.sourceforge.net/`
Java	PaRTeG	2011	`http://parteg.sourceforge.net/`
	Jester	2001	`http://jester.sourceforge.net/`
	μJava	2002	`http://www.isse.gmu.edu/~ofut/muJava/`
	Lava	2004	`http://igor.gold.ac`
	Jumble	2012	`http://sourceforge.net/projects/jumble/`
Python	Pester	2001	`http://jester.sourceforge.net/`

Table 8.16 is a partial list of mutation tools. Some of the tools listed here are now obsolete and replaced by newer versions while others continue to be used or are available for use. Mothra, Proteum, Proteum/IM, μJava, and Lava belong to the latter category. Lava is novel in the sense that it allows the generation of second order mutants; certainly Mothra has also been used in some scientific experiments to generate higher order mutants.

Milu is a new addition to the list of tools for mutation testing. It mutates C programs. It supports both first order and higher order mutations. MILU implements the C mutation operator set designed at Purdue University and summarized in Section 8.10.

Most mutation tools provide support for selecting a subset of the mutation operators to be applied and also the segment of the program under test. These two features allow a tester to perform incremental testing. Test

sets could be evaluated against critical components of applications. For example, in a nuclear plant control software, components that relate to the emergency shutdown system could be mutated by a larger number of mutation operators for test assessment.

While mutation testing is preferably done incrementally, it should also be done after the test set to be evaluated has been evaluated against the control flow and data flow metrics. Empirical studies have shown that a test data that is adequate with respect to the traditional control flow criteria, e.g. decision coverage, are not adequate with respect to mutation. Thus there are at least two advantages of applying mutation after establishing control flow adequacy. One, several mutants will likely be distinguished by control-flow adequate tests. Second, the mutants that remain live, will likely help enhance the test set.

In practice, data flow based test adequacy criteria turn out to be stronger than those based on control flow. Again, empirical studies have shown that test sets adequate with respect to the all-uses criterion are not adequate with respect to mutants generated by the ABS and ROR operators. Certainly, this is not true in some cases. However, this suggests that one could apply the ABS and ROR operators to evaluate the adequacy of a test set that is found to be adequate with respect to the all-uses criterion. By doing so, a tester will likely gain the two advantages mentioned above.

SUMMARY

This chapter offers a detailed introduction to mutation testing. While mutation offers a powerful technique for software testing, for some the technique might be difficult to understand, and hence the details. We have described in detail a test adequacy assessment procedure using mutation. Once this procedure is understood, it should not be difficult for a tester to invent variants to suit the individual process needs.

A novice to mutation testing might go away believing that mutation testing is intended to detect simple mistakes made by programmers. While detection of simple mistakes is certainly an intention of mutation testing, it constitutes a small portion of the overall benefit one will likely attain. Many independently conducted experiments over decades have revealed that mutation testing is highly capable of detecting complex faults that creep into programs. One example and several exercises are designed to show how mutants created by simple syntactic changes lead to the discovery of complex faults.

Mutation operators are an integral part of mutation testing. It is important for a tester to understand how such operators are designed and how they function. A basic understanding of

mutation operators, and experience in using them, helps a tester understand the purpose and strength of each operator. Further, such understanding opens the door for the construction of a completely new class of operators should one need to test an application written in a language for which no mutation operators have ever been designed! It is for these reasons that we have spent a substantial portion of this chapter presenting a complete set of mutation operators for C and Java.

Several strategies have been proposed by researchers to reduce the cost of mutation testing. Some of these are implemented in mutation tools while others serve as recommendations for testers. It is important to understand these strategies to avoid getting overwhelmed by mutation. Certainly, regardless of the strategy one employs, obtaining a high mutation score is often much more expensive than, for example, obtaining 100% decision coverage. However, the additional expense is highly likely to lead to improved reliability of the resulting product.

Exercises

8.1 Given program P in programming language L and a non-empty set M of mutation operators, what is the smallest number of mutants generated when mutation operators in M are applied to P?

8.2 (a) Let D and R denote, respectively, the sets of elements in the domain of mutation operator M. Suppose that program P contains exactly one occurrence of each element of D and that $D = R$. How many mutants of P will be generated by applying M? (b) How might your answer to (a) change if $D \neq R$?

8.3 Create at least two mutants of P8.2 that could lead to an incorrect signal sent to the caller of procedure `checkTemp`. Assume that you are allowed to create mutants by only changing constants, variables, arithmetic operators, and logical operators.

8.4 In Program P8.2, we have used three `if` statements to set `danger` to its appropriate value. Instead, suppose we use `if-else` structure as follows:

```
if (highCount==1) danger=moderate;
else if (highCount==2) danger=high;
else if (highCount==3) danger=veryHigh;
return(danger);
```

Consider mutant M3 of the revised program obtained by replacing `veryHigh` by `none`. Would the behavior of M3 differ from that of M1 and M2 in Examples 8.4 and 8.5?

8.5 In Example 8.12 the tests are selected in the order t_1, t_2, t_3, t_4. Suppose the tests are selected in the order t_4, t_3, t_2, t_1. By how much will this revised order change the number of mutant executions as compared to the number of executions in Example 8.12 ?

8.6 Why is it considered impossible to construct an ideal set of mutation operators for non-trivial programming languages? Does there exist a language for which such a set can be constructed?

8.7 Consider the statement: "In mutation testing, an assumption is that programmers make small errors." Critique this statement in the light of the Competent Programmer Hypothesis and the Coupling Effect.

8.8 Consider Program P8.3 in Example 8.17. Consider three mutants of 8.17 obtained as follows:

M_1: At line 4, replace `index < size` by `index < size − 1`.
M_2: At line 5, replace `atWeight > w` by `atWeight ≥ w`.
M_3: At line 5, replace `atWeight > w` by `atWeight > abs(w)`.

For each of the three mutants, derive a test case that distinguishes it from the parent or prove that it is an equivalent mutant.

8.9 Formulate a "Malicious programmer hypothesis" analogous to the Competant Programmer Hypothesis. In what domain of software testing do you think such a hypothesis might be useful?

8.10 Under weak mutation, a mutant is considered distinguished by a test case t if it satisfies conditions C_1 and C_2 in Section 8.3.4. Derive a minimal sized test set T for Program 8.3 that is adequate with respect to weak mutation given the three mutants in Exercise 8.8. Is T adequate with respect to strong mutation?

8.11 As explained in Section 8.3.2, adequacy assessment and test enhancement could be done concurrently by two or more testers. While the testers share a common test database, it might be possible that two or more testers derive tests that are redundant in the sense that a test already derived a by a tester might have distinguished a mutant that some other tester has not been able to distinguish.
Given that adequacy assessment is done by two testers, sketch a sequence of events that reveal the possibility of a redundant test case being generated. You could include events such as: Tester 1 generates mutants, tester 2 executes mutants, Tester 1 adds a new test case to the database, Tester 2 removes a test case from the database, and so on. Assume that (a) tester *TS* is allowed to remove a test case from the data base but only if it was created by *TS* and (b) the adequacy assessment process is started all over when a fault is discovered and fixed.

8.12 Consider naming the statement deletion operator for C as: *statement replacement by null statement*. Why would this be a poor naming choice?

8.13 Consider the following simple function.

```
1   function xRaisedToy(int x, y){
2    int power=1; count=y;
3    while(count>0){
4     power=power* x;
5     count=count-1;
6    }
7    return(power)
8   }
```

(a) How many *abs* mutants will be generated when `xRaisedToy` is mutated by the *abs* operator? (b) List at least two *abs* mutants generated? (c) Consider the following mutant of `xRaisedToy` obtained by mutating line 4:

`power=power*`*abs*`(x);`

Generate a test case that distinguishes this mutant, or show that it is equivalent to `xRaisedToy`.
(d) Generate at least one *abs* mutant equivalent to `xRaisedToy`.

8.14 Let T_D denote a decision-adequate test set for the function in Exercise 8.13. Show that there exists a T_D that fails to distinguish the *abs* mutant in part (c).

8.15 The AMC operator was designed to model faults in Java programs related to the incorrect choice of the access modifiers such as `private` and `public`. However, this operator was not implemented in the μJava system. Offer arguments in support of, or against, the decision not to include AMC in μJava.

8.16 Let P_J denote a Java method corresponding to function `xRaisedToy` in Exercise 8.13. Which Java mutation operators will generate a non-zero number of mutants for P_J given that no other part of the program containing P_J is to be mutated?

8.17 Derive conditions a test case must satisfy to distinguish each mutant shown in Example 8.44.

8.18 Given the following class declarations, list all mutants generated by applying the IOP operator:

```
class planet{
 String name;
 Orbit orbit ( . . . ){
 oType=high;
 ⋮
}
 ⋮
}
```

```
class farPlanet extends planet{
 float dist; Orbit orbit ( . . . );
 ⋮
 float currentDistance( String pName){
  oType=low; super.orbit( );
  dist=computeDistance(pName);
  return dist;
 } ⋮
}
```

8.19 Consider the following simple program P that contains an error at line 3. A mutant M, obtained by replacing this line as shown, is the correct program. The mutant operator used here replaces a variable by a constant from within the program. In C, this operator is named CRCR. Thus the error in the program is modeled by a mutation operator.

```
1   input x, y
2    if x < y then
3     z=x*(y+x)          ← z= x*(y+1)
4    else
5     z=x*(y−1)
```

(a) Construct a test case t_1 that causes P to fail and establishes the relation $P(t_1) \neq M(t_1)$.
(b) Can you construct a test case t_2 on which P is successful but $P(t_2) \neq M(t_2)$ does not hold?
(c) Construct a test set T that is adequate with respect to decision coverage, MC/DC, and all-uses coverage criteria and that does not reveal the error. Does any test in T distinguish M from P?

8.20 As reported by Donald Knuth, missing assignments accounted for over 11% of all the faults found in TEX. Considering a missing initialization as a special case of missing assignment, these faults account for over 15% of all faults found in TEX. This exercise illustrates how a missing assignment is detected using mutation. In the program listed next, the position of a missing assignment is indicated. Let P_c denote the correct program.

```
1   int x, p, q
2          ← Default value assignment,   x=0, is missing here.
3   if (q < 0 )x=p* q
4   if (p ≥ 0 ∧ q>0) x=p/q
4   y=x+1
```

Show that each of the following mutants, obtained by mutating line 3, is error revealing.

$$\texttt{if}(\texttt{q} < 1)\ \texttt{x} = \texttt{p} * \texttt{q}$$
$$\texttt{if}((\texttt{p} \geq -1) \wedge (\texttt{q} > 0))\ \texttt{x} = \texttt{p}/\texttt{q}$$
$$\texttt{if}((\texttt{p} \geq 1) \wedge (\texttt{q} > 0))\ \texttt{x} = \texttt{p}/\texttt{q}$$
$$\texttt{if}((\texttt{p} \geq 0) \wedge (\texttt{q} > -1))\ \texttt{x} = \texttt{p}/\texttt{q}$$
$$\texttt{if}((\texttt{p} \geq 0) \wedge (\texttt{q} > 1))\ \texttt{x} = \texttt{p}/\texttt{q}$$

One might argue that the missing initialization above can be detected easily through static analysis. Assuming that the program fragment above is a small portion of a large program, can you argue that static analysis might fail to detect the missing initialization?

8.21 Example 8.21 illustrates how a simple mutation can reveal a missing condition error. The example showed that the error must be revealed when mutation adequacy has been achieved. Develop test sets T_1, T_2, and T_3 adequate with respect to, respectively, decision coverage, MC/DC, and all-uses coverage criteria such that none of the tests reveals the error in function `misCond`?

8.22 Some program faults might escape detection using mutation as well as any path oriented testing strategy. Dick Hamlet provides a simple example of one such error. Hamlet's program is reproduced below.

```
real hamletFunction (x: real){
 if (x< 0) x= -x;
 return(sqr(x)*2)        ← This should be return(sqrt(x) * 2)
}
```

The error is in the use of function *sqr*, read as "square," instead of the function *sqrt*, read as "square root." Notice that there is a simple mutation that exists in this program, that of *sqrt* being mutated by the programmer to *sqr*. However, let us assume that there is no such mutation operator in the mutation tool that you are using. (Of course, in principle, there is nothing which prevents a mutation testing tool from providing such a mutation operator.) Hence, the call to *sqr* will not be mutated to *sqrt*.

Consider the test set $T = \{< \texttt{x} = 0 >, < \texttt{x} = -1 >, < \texttt{x} = 1 >\}$. It is easy to check that `hamletFunction` behaves correctly on T. Show that T is adequate with respect to (a) all path oriented coverage criteria, such as all-uses coverage and MC/DC, and (b) all mutants obtained by using the mutation operators in Table 8.14. It would be best to complete this exercise using a mutation tool. If necessary, translate `hamletFunction` into C if you have access to Proteum or to Java if you have access to μJava.

8.23 Let M be a mutant of program P generated by applying some mutation operator. Suppose that M is equivalent to P. Despite the equivalence, M might be a useful program, and perform better (or worse) than P. Examine Hoare's FIND program and its mutant obtained by replacing the

conditionI.GT.F, labeled 70, by FALSE. Is the mutant equivalent to its parent? Does it perform better or worse than the original program?

8.24 Let $MS(X)$ denote the mutation score of test set X with respect to some program and a set of mutation operators. Let T_A be a test set for application A and $U_A \subseteq T_A$. Develop an efficient minimization algorithm to construct $T'_A = T_A - U_A$ such that (a) $MS(T_A) = MS(T'_A)$ and (b) for any $T''_A \subset T_A$, $|T'_A| < |T''_A|$. (Note: Such minimization algorithms are useful in the removal of tests that are redundant with respect to the mutation adequacy criterion. One could construct similar algorithms for minimization of a test set with respect to any other quantitative adequacy criterion.)

8.25 Suppose a class C contains at an overloaded method with three versions. What is the minimum number of mutants generated when the OAN operator is applied to C?

8.26 The following are believed to be "myths of mutation." Do you agree?

a. It is an error seeding approach.
b. It is too expensive relative to other test adequacy criteria.
c. It is "harder" than other coverage based test methods in the sense that it requires a tester to think much more while developing a mutation adequate test.
d. It discovers only simple errors.
e. It is applicable only during unit testing.
f. It is applicable only to numerical programs, e.g. a program that inverts a matrix.

8.27 Consider the following program originally due to Jeff Voas, Larry Morell, and Keith Miller.

```
bool noSolution=false;
int vmmFunction (a, b, c: int){
 real  x;
 if (a≠ 0){
  d=b*b-5*a*c;      ← Constant 5 should be 4.
  if(d<0)
   x=0;
  else
   x=(−b+trunc(sqrt(d)));
 }
 else
  x=−c div b;
 if(a*x*x+b*x+c==0)
  return(x);
 else{
  noSolution=true;
  return(0);
 }
}
```

Given inputs a, b, and c, `vmmFunction` is required to find an integral solution to the quadratic equation $a * x^2 + b * x + c = 0$. If an integral solution exists, the program returns with the solution, if not then it sets the global variable `noSolution` to `true` and returns a 0. Function `vmmFunction` has a fault at line 5, constant 5 should instead be 4.

a. Consider the set M_W of mutants created by applying the mutation operators suggested by Wong, as in Table 8.15, to `vmmFunction`. Is any mutant in M_W error revealing?
b. Now consider the set M_O of mutants created by applying the mutation operators suggested by Offutt, as in Table 8.15, to `vmmFunction`. Is any mutant in M_O error revealing?
c. Next, suppose that we have fixed the fault at line 5 but line 6 now contains a fault, the faulty condition being `d` ≤ 0. Is there any error revealing mutant of `vmmFunction` in M_W? M_O? (Note that M_W and M_O are now created by applying the mutation operators to `vmmFunction` with the fault in the condition at line 6.)

It might be best to do this exercise with the help of a tool such as Proteum or μJava. When using a tool, you will need to translate the pseudo-code of `vmmFuntion` into C or Java, as appropriate.

8.28 Program P8.4 shows a function named `count` that is required to read a character and an integer value. If the character input is 'a' then the entry `a[value]` is incremented by one. If the character input is a 'b' then entry `b[value]` is incremented by 1. The input value is ignored for all other characters. The value input must satisfy the condition $0 \leq$ `value` $< N$.

The function `count` is supposed to read the input until it gets an end-of-file character, denoted by `eof`, when it terminates. Arrays `a` and `b` are global to the function. Note that as shown here, `count` does not check if the input value is in its intended range. Thus, an input integer value could force the function, and its container program, to misbehave.

Program P8.4

```
1   #define N=10
2   int a[N], b[N];
3   void count( ){
4   char code; int value;
5    for (i=1; i≤ N;i++){
6     a[i]=0;
7     b[i]=0;
8    }
9    while (¬ eof){
10   input(code, value);
11   if (code=='a')a[value]=a[value]+1;
12    elseif (code=='b') b[value]=b[value]+1;
13   }
14  }
```

(a) Is it possible to develop a test set that is adequate with respect to the MC/DC criterion such that the input integer value is within its intended range in each test case?
(b) Is it possible to develop a test set that is adequate with respect to the all-uses coverage criterion such that the input integer value is within its intended range in each test case?
(c) Is it possible to develop a test set that is adequate with respect to the mutation criterion such that the input integer value is within its intended range in each test case? Assume that all traditional mutation operators listed in Table 8.14 are used.

8.29 Some programming languages provide the set type. Java provides a SET interface. Specification language Z, functional language Miranda, and procedural language SETL are examples of languages that provide the set type. Consider a mutation operator named SM–abbreviation for set mutation. SM is applied to any object of type set in a specification. Thus, the application of SM to specification S_1 mutates it to another specification S_2 by changing the contents of an object defined as a set.

(a) Develop a suitable semantics for SM. Note that traditional mutation operators such as variable replacement will replace an object x of type set with another object y defined in the same program. However, SM is different in that it mutates the value of a set object in a specification.
(b) Think of applying SM to a program during execution by mutating the value of a set object when it is referenced. Why would such a mutation be useful?

This page is intentionally left blank.

Part IV
Phases of Testing

Software testing pervades the entire software development process. This part of the book focuses on three phases of testing: unit testing, regression testing, and integration testing. A significant challenge in regression testing is the selection of tests to rerun. Methods for the selection of a subset of tests to rerun are covered in Chapter 9. Unit testing is covered in Chapter 10. Unit testing is used widely in the industry. Here the focus is on tools for unit testing and some techniques, e.g., the use of mock objects, that make unit testing easier.

Deciding a suitable sequence in which components of a system are to be integrated is the *test order problem*. There exist several solutions to this problem. Some of these are discussed in Chapter 11. This chapter also introduces a variety of terminology that is often used in integration testing.

Tasks performed in other phases of testing including unit testing and system testing are covered across several chapters. These tasks include test generation and adequacy assessment. Tools useful in these phases are also sampled in other chapters.

This page is intentionally left blank.

9

Test Selection, Minimization, and Prioritization for Regression Testing

CONTENTS

9.1 What is regression testing?
9.2 Regression test process
9.3 Regression test selection: the problem
9.4 Selecting regression tests
9.5 Test selection using execution trace
9.6 Test selection using dynamic slicing
9.7 Scalability of test selection algorithms
9.8 Test minimization
9.9 Test prioritization
9.10 Tools

The purpose of this chapter is to introduce techniques for the selection, minimization, and prioritization of tests for regression testing. The source T from which tests are selected is likely derived using a combination of black-box and white-box techniques and used for system or component testing. However, when this system or component is modified, for whatever reason, one might be able to retest it using only a subset of T and ensure that despite the changes, the existing unchanged code continues to function as desired. A sample of techniques for the selection, minimization, and prioritization of this subset are presented in this chapter.

9.1 What Is Regression Testing?

The word *regress* means to return to a previous, usually worse, state. Regression testing refers to that portion of the test cycle in which a program P' is tested to ensure that not only does the newly added or modified code behaves correctly, but also that code carried over unchanged from the previous version P continues to behave correctly. Thus regression testing is useful, and needed, whenever a new version of a program is obtained by modifying an existing version.

A program that is modified for reasons such as correction, addition of features, etc., is often retested to ensure that the unchanged parts of the program continue to work correctly. Such testing is commonly referred to as "regression testing."

Regression testing is sometimes referred to as "program revalidation." The term "corrective regression testing" refers to regression testing of a program obtained by making corrections to the previous versions. Another term "progressive regression testing" refers to regression testing of a program obtained by adding new features. A typical regression testing scenario often includes both corrective and progressive regression testing. In any case, techniques described in this chapter are applicable to both types of regression testing.

To understand the process of regression testing, let us examine a development cycle exhibited in Figure 9.1. The figure shows a highly simplified develop-test-release process for program P, referred to as Version 1. While P is in use, there might be a need to add new features, remove any reported errors, and rewrite some code to improve performance. Such modifications lead to P', referred to as Version 2. This modified version must be tested for any new functionality (step 5 in the figure). However, when making modifications to P the developers might mistakenly add or remove code that causes the existing and unchanged functionality from P to stop

Version 1	Version 2
1. Develop P	4. Modify P to P'
2. Test P	5. Test P' for new functionality
3. Release P	6. Perform regression testing on P' to ensure that the code carried over from P behaves correctly
	7. Release P'

Figure 9.1 Two phases of product development and maintenance. Version 1 (P) is developed, tested, and released in the first phase. In the next phase, Version 2 (P') is obtained by modifying Version 1.

behaving as desired. One performs regression testing (step 6) to ensure that any malfunction of the existing code is detected and repaired prior to the release of P'.

It should be obvious from the above description that regression testing can be applied in each phase of software development. For example, during unit testing, when a given unit such as a class is modified by adding new methods, one needs to perform regression testing to ensure that methods not modified continue to work as required. Certainly, regression testing is redundant in cases where the developer can demonstrate through suitable arguments that the methods added can have no effect on the existing methods.

Regression testing may occur in any phase of software development. Thus, it might not be a distinct test phase by itself.

Regression testing is also needed when a subsystem is modified to generate a new version of an application. When one or more components of an application are modified, the entire application must also be subject to regression testing. In some cases regression testing might be needed when the underlying hardware changes. In this case, regression testing is performed despite any change in the software.

In the remainder of this chapter you will find various techniques for regression testing. It is important to note that some techniques introduced in this chapter, while sophisticated, might not be applicable in certain environments while absolutely necessary in others. Hence, it is important to understand not only the technique for regression testing but also its strengths and limitations.

9.2 Regression Test Process

A regression test process is exhibited in Figure 9.2. The process assumes that P' is available for regression testing. There is usually a long series of tasks that lead to P' from P. These tasks, not shown in Figure 9.2, include creation of one or more modification requests and the actual modification of the design and the code. A modification request might lead to a simple error fix, or to a complex redesign and coding of a component of P. In any case, regression testing is recommended after P has been modified and any newly added functionality tested and found correct.

Regression testing becomes necessary after a program is changed due to modification requests. Such testing ensures that the changes made do not affect the correctness of other parts of the program.

The tasks in Figure 9.2 are shown as if they occur in the given sequence. This is not necessarily true and other sequencings are possible. Several of the tasks shown can be completed while P is being modified to P'. It is important to note that except in some cases, for test selection, all tasks shown in the figure are performed in almost all phases of testing and are not specific to regression testing.

9.2.1 Revalidation, selection, minimization, and prioritization

A brute force regression test might run all tests for the original program P against the modified program P'. However, for several reasons, doing so might not be feasible or desirable.

While it would be ideal to test P' against all tests developed for P, this might not be possible for several reasons. For example, there might not be sufficient time available to run all tests. Also, some tests for P might become invalid for P' due to reasons such as a change in the input data and its format for one or more features. In yet another scenario, the inputs specified in some tests might remain valid for P' but the expected output might not. These are some reasons that necessitate step 1 in Figure 9.2.

Prior to executing a modified program against the test cases used in its previous incarnation, one needs to ensure that these tests are valid. A valid test is one that will test, as desired, a feature in the modified program.

Test revalidation refers to the task of checking which tests for P remain valid for P'. Revalidation is necessary to ensure that only tests that are applicable to P' are used during regression testing.

Test selection can be interpreted in several ways. Validated tests might be redundant in that they do not traverse any of the modified portions in P'. The identification of tests that traverse modified portions of P' is often referred to as test selection and sometimes as the *regression test selection* (RTS) problem. However, note that both test minimization and prioritization described next are also techniques for test selection.

The regression test selection problem is to identify tests to be run for regression testing of a modified program. Test minimization and test prioritization are two techniques for regression test selection.

Test minimization discards tests seemingly redundant with respect to some criteria. For example, if both t_1 and t_2 test function f in P then one might decide to discard t_2 in favor of t_1. The purpose of minimization is to reduce the number of tests to execute for regression testing.

Test prioritization refers to the task of prioritizing tests based on some criteria. A set of prioritized tests becomes useful when only a subset of tests can be executed due to resource constraints. Test selection can be achieved by selecting a few tests from a prioritized list. However, several other methods for test selection are available as discussed later in this chapter. Revalidation, followed by selection, minimization, and prioritization is one possible sequence to execute these tasks.

Figure 9.2 A subset of tasks in regression testing.

Example 9.1 A Web service is a program that can be used by another program over the Web. Consider a Web service named ZC, short for ZipCode. The initial version of ZC provides two services: ZtoC and ZtoA. Service ZtoC inputs a zip code and returns a list of cities and the corresponding state while ZtoA inputs a zip code and returns the corresponding area code. We assume that while the ZipCode service can be used over the Web from wherever an Internet connection is available, it serves only the United States.

Let us suppose that ZC has been modified to ZC' as follows. First, a user can select from a list of countries and supply the zip code to obtain the corresponding city in that country. This modification is made only to the ZtoC function while ZtoA remains unchanged. Note that the term "zip code" is not universal. For example, in India, the equivalent term is "pin code" which is 6-digits long as compared to the 5-digit zip code used in the United States. Second, a new service named ZtoT has been added which inputs a country and a zip code and returns the corresponding time zone.

Consider the following two tests (only inputs specified) used for testing ZC:

t_1: $< service = ZtoC, zip = 47906 >$
t_2: $< service = ZtoA, zip = 47906 >$

A simple examination of the two tests reveals that test t_1 is not valid for ZC' as it does not list the required country field. Test t_2 is valid as we have made no change to ZtoA. Thus we need to either discard t_1 and replace it by a new test for the modified ZtoC or simply modify t_1 appropriately. We prefer to modify and hence our validated regression test suite for ZC' is

t_1: $< country = USA, service = ZtoC, zip = 47906 >$
t_2: $< service = ZtoA, zip = 47906 >$

Note that testing ZC' requires additional tests to test the ZtoT service. However, we need only the two tests listed above for regression testing. To keep this example short, we have listed only a few tests for ZC. In practice one would develop a much larger suite of tests for ZC which will then be the source of regression tests for ZC'.

9.2.2 Test setup

Test setup refers to the process by which the application under test is placed in its intended, or simulated, environment ready to receive data and able to transfer any desired output information. This process could be as simple

as double clicking on the application icon to launch it for testing and as complex as setting up the entire special purpose hardware and monitoring equipment and initializing the environment before the test could begin. Test setup becomes even more challenging when testing embedded software such as that found in printers, cell phones, Automated Teller Machines, medical devices, and automobile engine controllers.

Note that test setup is not special to regression testing, it is also necessary during other stages of testing such as during integration or system testing. Often test setup requires the use of simulators that allow the replacement of a "real device" to be controlled by the software with its simulated version. For example, a heart simulator is used while testing a commonly used heart control device known as the pacemaker. The simulator allows the pacemaker software to be tested without having to install it inside a human body.

The test setup process and the setup itself are highly dependent on the application under test and its hardware and software environment. For example, the test setup process and the setup for an automobile engine control software is quite different from that of a cell phone. In the former one needs an engine simulator, or the actual automobile engine to be controlled, while in the latter one needs a test driver that can simulate the constantly changing environment.

9.2.3 Test sequencing

The sequence in which a program is executed against tests becomes important when the state of the program and that of its environment determines the program's response to the next test case. Banking applications are one example where test sequencing becomes important.

The sequence in which tests are input to an application may or may not be of concern. Test sequencing often becomes important for an application that has an internal state and is continuously running. Banking software, Web service, engine controller are examples of such applications. Sequencing requires grouping and sequencing tests to be run together. The following example illustrates the importance of test sequencing.

Example 9.2 Consider a simplified banking application referred to as SATM. Application SATM maintains account balances and offers users the following functionality: login, deposit, withdraw, and exit. Data for each account are maintained in a secure database.

Figure 9.3 exhibits the behavior of SATM as a finite state machine. Note that the machine has six distinct states, some referred to as modes in the figure. These are labeled as Initialize, LM, RM, DM, UM, and WM. When launched, the SATM performs initialization operations, generates an "ID?" message, and moves to the LM state. If a user enters a valid ID, SATM moves to the RM state else it remains in the LM state and again requests for an ID.

Figure 9.3 State transition in a simplified banking application. Transitions are labeled as X/Y, where X indicates an input and Y the expected output. "Complete" is an internal input indicating that the application moves to the next state upon completion of operations in its current state.

While in the RM state the application expects a service request. Upon receiving a Deposit request it enters the DM state and asks for an amount to be deposited. Upon receiving an amount it generates a confirmatory message and moves to the UM state where it updates the account balance and gets back to the RM state. A similar behavior is shown for the Withdraw request. SATM exits the RM state upon receiving an Exit request.

As an example, the state transitions in a banking application require that tests be sequenced appropriately.

Let us now consider a set of three tests designed to test the Login, Deposit, Withdraw, and Exit features of SATM. The tests are given in the following table in the form of a test matrix. Each test requires that the application be launched fresh and the user (tester in this case) log in. We assume that the user with ID=1 begins with an account balance of 0. Test t_1 checks the login module and the Exit feature, t_2 the Deposit module, and t_3 the Withdraw module. As you

Test	Input sequence	Expected output sequence	Module tested
t_1	ID=1, Request=Exit	Welcome, Bye	Login
t_2	ID=1, Request=Deposit, Amount=50	D? Welcome, Amount? OK, Done, 50	Deposit
t_3	ID=1, Request= Withdraw, Amount=30	ID? Welcome, Amount? 30, Done, 20	Withdraw

might have guessed, these tests are not sufficient for a thorough test of SATM, but they suffice to illustrate the need for test sequencing as explained next.

Now suppose that the Withdraw module has been modified to implement a change in withdrawal policy, e.g. "No more than $300 can be withdrawn on any single day." We now have the modified SATM' to be tested for the new functionality as well as to check if none of the existing functionality has broken. What tests should be rerun?

Assuming that no other module of SATM has been modified, one might propose that tests t_1 and t_2 need not be rerun. This is a risky proposition unless some formal technique is used to prove that indeed the changes made to the Withdraw module cannot affect the behavior of the remaining modules.

Let us assume that the testers are convinced that the changes in SATM will not affect any module other than Withdraw. Does this mean that we can run only t_3 as a regression test? The answer is in the negative. Recall our assumption that testing of SATM begins with an account balance of 0 for the user with ID=1. Under this assumption, when run as the first test, t_3 will likely fail because the expected output will not match the output generated by SATM' (see Exercise 9.1).

The argument above leads us to conclude that we need to run test t_3 after having run t_2. Running t_2 ensures that SATM' is brought to the state in which we expect test t_3 to be successful.

Note that the finite state machine shown in Figure 9.3 ignores the values of internal variables and databases used by SATM and SATM'. During regression as well as many other types of testing, test sequencing is often necessary to bring the application to a state where the values of internal variables, and contents of the databases used, correspond to the intention at the time of designing the tests. It is advisable that such intentions (or assumptions) be documented along with each test.

Regression tests can be executed automatically using some of the existing tools, also known as "test runners." Nevertheless, no general purpose tool might be applicable in a given environment for the automated execution of regression tests.

9.2.4 Test execution

Once the testing infrastructure has been set up, tests selected, revalidated, and sequenced, it is time to execute them. This task is often automated using a generic or a special purpose tool. General purpose tools are available to run regression tests for applications such as Web service (see Section 9.10). However, most embedded systems, due to their unique hardware requirements, often require special purpose tools that

input a test suite and automatically run the application against it in a batch mode.

The importance of a tool for test execution cannot be overemphasized. Commercial applications tend to be large and the size of the regression test suite usually increases as new versions arrive. Manual execution of regression tests might become impractical and error prone.

Automated testing requires running a program against inputs and checking if the output is correct. The latter is the "oracle" problem. In the case of regression testing the expected test outputs might be available from the past.

9.2.5 Output comparison

Each test needs verification. This is also done automatically with the help of the test execution tool that compares the generated output with the expected output. However, this might not be a simple process, especially in embedded systems. In such systems, often it is the internal state of the application, or the state of the hardware controlled by the application, that must be checked. This is one reason why generic tools that offer an oracle might not be appropriate for test verification.

One of the goals for test execution is to measure an application's performance. For example, one might want to know how many requests per second can be processed by a Web service. In this case performance, and not functional correctness, is of interest. The test execution tool must have special features to allow such measurements.

9.3 Regression Test Selection: The Problem

Let us examine the regression testing problem with respect to Figure 9.4. Let P denote Version X that has been tested using test set T against specification S. Let P' be generated by modifying P. The behavior of P' must conform to specification S'. Specifications S and S' could be the same and P' is the result of modifying P to remove faults. S' could also be different from S in that S' contains all features in S and a few more, or that one of the features in S has been redefined in S'.

The regression test selection problem is to select a set of tests T such that the execution of the modified program P' against T will ensure that the functionality carried over from P continues to work correctly.

The regression testing problem is to find a test set T_r on which P' is to be tested to ensure that code that implements functionality carried over from P works correctly. As shown in Figure 9.4, often T_r is a subset of T used for testing P.

In addition to regression testing, P' must also be tested to ensure that the newly added code behaves correctly. This is done using a newly developed test set T_d. Thus P' is tested against $T' = T_r \cup T_d$ where T_r is the regression test suite and T_d the development test suite intended to test any new functionality in P'. Note that we have subdivided T into three categories: redundant tests (T_u), obsolete tests (T_o), and regression tests (T_r). While P

The modified program P must also be tested to ensure the newly implemented code is correct.

Some tests from the previous version of the modified program might become obsolete and will not be included in the regression tests for P'.

Figure 9.4 Regressing testing as a test selection problem. A subset T_r of set T is selected for retesting the functionality of P that remains unchanged in P'.

is executed against the entire T, P' is executed only against the regression set T_r and the development set T_d. Tests in T that cause P to terminate prematurely or enter into an infinite loop might be included in T_o or in T_r depending on their purpose.

In summary, the regression test selection problem (RTS) problem is stated as follows: Find a minimal T_r such that $\forall t \in T_r$ and $t' \in T_u \cup T_r, P(t) = P'(t) \implies P(t') = P'(t')$. In other words, the RTS problem is to find a minimal subset T_r of non-obsolete tests from T such that if P' passes tests in T_r then it will also pass tests in T_u. Notice that determination of T_r requires that we know the set of obsolete tests T_o. Obsolete tests are those no longer valid for P' for some reason.

Identification of obsolete tests is likely to be a manual activity.

Identification of obsolete tests is a largely manual activity. As mentioned earlier, this activity is often referred to as *test case revalidation*. A test case valid for P might be invalid for P' because the input, output, or both input and output components of the test are rendered invalid by the modification to P. Such a test case either becomes obsolete and is discarded while testing P' or is corrected and becomes a part of T_r or T_u.

Note that the notion of "correctness" in the above discussion is with respect to the correct functional behavior. It is possible that a solution to the RTS problem ignores a test case t on which P' fails to meet its performance requirement whereas P does. Algorithms for test selection in the remainder of this chapter ignore the performance requirement.

9.4 Selecting Regression Tests

In this section, we summarize several techniques for selecting tests for regression testing. Details of some of these techniques follow in the subsequent sections.

9.4.1 Test all

This is perhaps the simplest of all regression testing techniques. The tester is unwilling to take any risks and tests the modified program P' against all non-obsolete tests from P. Thus, with reference to Figure 9.4, we use $T' = T - T_o$ in the test-all strategy.

A straightforward way to select regression tests is to select them all!

While the test-all strategy might be the least risky among the ones described here, it does suffer from a significant disadvantage. Suppose that P' has been developed and the added functionality verified. Assume that one week remains to release P'. However, the test-all strategy will require at least 3-weeks for completion. In this situation one might want to use a smaller regression test suite than $T - T_o$. Various techniques for obtaining smaller regression test sets are discussed later in this chapter. Nevertheless, the test-all strategy combined with tools for automated regression test tools is perhaps the most widely used technique in the commercial world.

9.4.2 Random selection

Random selection of tests for regression testing is one possible method to reduce the number of tests. In this approach tests are selected randomly from the set $T - T_o$. The tester can decide how many tests to select depending on the level of confidence required and the available time for regression testing.

Another way to select regression tests is to select them randomly from the full set of tests.

Under the assumption that all tests are equally good in their fault detection ability, the confidence in the correctness of the unchanged code increases with an increase in the number of tests sampled and executed successfully. However, in most practical applications such an assumption is unlikely to hold. This is because some of the sampled tests might bear no relationship to the modified code while others might. This is the prime weakness of the random selection approach to regression testing. Nevertheless, random selection might turn out to be better than no regression testing at all.

9.4.3 Selecting modification traversing tests

Several regression test selection techniques aim to select a subset of T such that only tests that guarantee the execution of modified code and code that might be impacted by the modified code in P' are selected while those

Tests that traverse the code that has been modified are good candidates to select for regression testing.

that do not are discarded. These techniques use methods to determine the desired subset and aim at obtaining a minimal regression test suite. Techniques that obtain a minimal regression test suite without discarding any test that will traverse a modified statement are known as "safe" regression test selection techniques.

A key advantage of modification traversing tests is that when under time crunch, testers need to execute only a relatively smaller number of regression tests. Given that a "safe" technique is used, execution of such tests is likely a superior alternative to the test-all and random-selection strategies.

The sophistication of techniques to select modification traversing tests requires automation. It is impractical to apply these techniques to large commercial systems unless a tool is available that incorporates at least one safe test minimization technique. Further, while test selection appears attractive from the test effort point of view, it might not be a practical technique when tests are dependent on each other in complex ways and that this dependency cannot be incorporated in the test selection tool.

Two or more tests selected for regression might traverse exactly the same portions of the modified code. Some of these tests could be considered as redundant and excluded from the regression test. Doing so reduces the size of the regression test suite and the time to complete the test.

9.4.4 Test minimization

Suppose that T_r is a modification traversing subset of T. There are techniques that could further reduce the size of T_r. Such test minimization techniques aim at discarding redundant tests from T_r. A test t in T_r is considered redundant if another test u in T_r achieves the same objective as t. The objective in this context is often specified in terms of code coverage such as basic block coverage or any other form of control flow or data flow coverage. Requirements coverage is another possible objective used for test minimization.

While test minimization might lead to a significant reduction in the size of the regression test suite, it is not necessarily safe. When tests are designed carefully, minimization might discard a test case whose objective does match that used for minimization. The next example illustrates this point.

Test minimization might not be safe in the sense that tests not selected might be actually useful in revealing errors despite the fact that they are considered redundant according to some criterion (black- or white-box).

Example 9.3 Consider the following trivial program P' required to output the sum of two input integers. However, due to an error, the program outputs the difference of the two integers.

```
int x, y;
output (x−y)
```

Now suppose that T_r contains 10 tests, nine with $y = 0$ and one, say t_{nz}, with non-zero values of both x and y. t_{nz} is the only test that causes P' to fail.

Suppose that T_r is minimized so that the basic block coverage obtained by executing P' remains unchanged. Obviously, each of the 10 tests in T_r covers the lone basic block in P' and therefore all but one test will be discarded by a minimization algorithm. If t_{nz} is the one discarded then the error in P' will not be revealed by the minimized test suite.

While the above example might seem trivial, it does point to the weakness of test minimization. Situations depicted in this example could arise in different forms in realistic applications. Hence it is recommended that minimization be carried out with caution. Tests discarded by a minimization algorithm must be reviewed carefully before being truely discarded.

9.4.5 Test prioritization

One approach to regression test selection is through test prioritization. In this approach, a suitable metric is used to rank all tests in T_r (see Figure 9.4). A test with the highest rank has the highest priority, the one with the next highest rank has the second highest priority, and so on. Prioritization does not eliminate any test from T_r. Instead, it allows the tester to decide which tests to select based on their relative priority and individual relationships based on sequencing and other requirements.

Once tests are selected for regression testing, they could be prioritized (ranked) based on one or more criteria. A ranked list of tests can be used to decide when to stop testing in case not enough resources are available to run all tests.

Example 9.4 Let R_1, R_2, and R_3 be three requirements carried over unchanged from P to P'. One approach first ranks the requirements for P' according to their criticality. For example, a ranking might be: R_2 most critical, followed by R_3, and lastly R_1. Any new requirements implemented in P' but not in P are not used in this ranking as they correspond to new tests in set T_d as shown in Figure 9.4.

Now suppose that the regression subset T_r from P is $\{t_1, t_2, t_3, t_4, t_5\}$ and that t_1 tests R_1, t_2 and t_3 test R_2, and t_4 and t_5 test R_3. We can now prioritize the tests in this order: t_2, t_3, t_4, t_5, t_1, where t_2 has the highest priority for execution and t_1 the lowest. It is up to the tester to decide which tests to select for regression testing. If the tester believes that changes made to P to arrive at P' are unlikely to have any effect on code that implements R_3, then tests t_4 and t_5 need not be executed. In case the tester has resources to run only three tests before P' is to be released, then t_2, t_3, and t_4 can be selected.

Several other more sophisticated techniques are available for test prioritization. Some of these use the amount of code covered as a metric to prioritize tests. These techniques, and others for test selection, are discussed in the following sections.

9.5 Test Selection Using Execution Trace

An execution trace of a program P is a sequence of program elements executed when P is run against a test.

Let P be a program containing one or more functions. P has been tested against tests in T as shown in Figure 9.4. P is modified to P' by adding new functionality and fixing some known errors. The newly added functionality has been tested and found correct. Also, the corrections made have been tested and found adequate. Our goal now is to test P' to ensure that the changes made do not affect the functionality carried over from P. While this could be achieved by executing P' against all the non-obsolete tests in T, we want to select only those that are necessary to check if the modifications made do not affect functionality common to P and P'.

Our first technique for selecting a subset of T is based on the use of execution slice obtained from the execution trace of P. The technique can be split into two phases. In the first phase, P is executed and the execution slice recorded for each test case in $T_{no} = T_u \cup T_r$ (see Figure 9.4). T_{no} contains no obsolete test cases and hence is a candidate for full regression test. In the second phase, the modified program P' is compared with P and T_r is isolated from T_{no} by an analysis of the execution slice obtained in the first phase.

9.5.1 Obtaining the execution trace

Let $G = (N, E)$ denote the control flow graph (CFG) of P, N being a set of nodes and E a set of directed edges joining the nodes. Each node in N corresponds to a basic block in P. *Start* and *End* are two special elements of N such that *Start* has no ancestor and *End* has no descendent. For each function f in P we generate a separate CFG denoted by G_f. The CFG for P is also known as the main CFG and corresponds to the main function in P. All other CFGs are known as child CFGs. When necessary, we will use the notation CFG(f) to refer to the CFG of function f.

Nodes in each CFG are numbered as 1, 2, and so on with 1 being the number of the *Start* node. A node in a CFG is referred to by prefixing the function name to its number. For example, P.3 is node 3 in G_P and f.2 node 2 in G_f.

A possible execution trace for program P executed against a test is the sequence of nodes in the control flow graph of P that are touched during execution.

We execute P against each test in T_{no}. During execution against $t \in T_{no}$, the execution trace is recorded as *trace*(t). The execution trace is a sequence of nodes. We save an execution trace as a set of nodes touched during the execution of P. Such a set is also known as an execution slice of P. *Start* is the first node in an execution trace and *End* the last node. Notice that an execution trace contains nodes from functions of P invoked during its execution.

Figure 9.5 Control flow graph for function `main`, and its child functions `g1` and `g2` in Program 9.1. Line numbers corresponding to each basic block, represented as a node, are placed to the left of the corresponding node. Labels t and f indicate, respectively, true and false values of the condition in the corresponding node. Dotted lines indicate points of function calls.

Example 9.5 Consider Program 9.1. It contains three functions: `main`, `g1`, and `g2`. The CFGs and child CFGs appear in Figure 9.5.

Program P9.1

```
1  main(){
2  int x, y, p;
3  input (x, y);
4  if (x< y)
5    p=g1(x, y);
6  else
7    p=g2(x, y);
8  endif
9  output (p);
10 end
11 }
```

```
1  int g1(int a, b){
2  int a, b;
3  if(a+1==b)
4    return(a*a);
5  else
6    return(b*b);
7  }
```

```
1  int g2(int a, b){
2  int a, b;
3  if(a==(b+1))
4    return(b*b);
5  else
6    return(a*a);
7  }
```

Now consider the following test set:

$$T = \left\{ \begin{array}{ll} t_1: & <x=1, y=3> \\ t_2: & <x=2, y=1> \\ t_3: & <x=3, y=4> \end{array} \right\}$$

Executing Program P9.1 against the three test cases in T results in the following execution traces corresponding to each test and the CFGs in Figure 9.5. We have shown the execution trace as a sequence of nodes traversed. However, a tool that generates the trace can, for the purpose of test selection, save it as a set of nodes, thereby conserving memory space.

Test (t)	Execution trace ($trace(t)$)
t_1	main.Start, main.1, main.2, g1.Start, g1.1, g1.3, g1.End, main.2, main.4, main.End.
t_2	main.Start, main.1, main.3, g2.Start, g2.1, g2.2, g2.End, main.3, main.4, main.End.
t_3	main.Start, main.1, main.3, g2.Start, g2.1, g2.3, g2.End, main.2, main.3, main.End.

Finding an execution trace for a continuously running program might become challenging if the program needs to be brought to its initial state for each trace.

While collecting the execution trace we assume that P is started in its initial state for each test input. This might create a problem for continuously running programs, e.g. an embedded system that requires initial setup and then responds to external events that serve as test inputs. In such cases a sequence of external events serves as test case. Nevertheless, we assume that P is brought into an initial state prior to the application of any external sequence of inputs that is considered a test case in T_{no}.

Let $test(n)$ denote the set of tests such that each test in $test(n)$ traversed node n at least once. Given the execution trace for each test in T_{no}, it is easy to find $test(n)$ for each $n \in N$. $test(n)$ is also known as test vector corresponding to node n.

Example 9.6 Test vectors for each node in the CFGs shown in Figure 9.5 can be found from the execution traces given in Example 9.5. These are listed in the following table. All tests traverse the *Start* and *End* nodes and hence the corresponding test vectors are not listed.

Function	Test vector ($test(n)$) for node n			
	1	2	3	4
main	t_1, t_2, t_3	t_1, t_3	t_2	t_1, t_2, t_3
g1	t_1, t_3	t_3	t_1	–
g2	t_2	t_2	None	–

9.5.2 Selecting regression tests

This is the second phase in the selection of modification traversing regression tests. It begins when P' is ready, has been tested and found correct in

any added functionality and error fixes. The two key steps in this phase are: construct CFG and syntax trees for P' and select tests. These two steps are described next.

Construct CFG and syntax trees: The first step in this phase is to get the CFG of P' denoted as $G' = (N', E')$. We now have G and G' as the CFGs of P and P'. Recall that other than the special nodes, each node in a CFG corresponds to a basic block.

Selection of tests using execution traces requires the use of control flow graphs of the original program P and the modified version P'. Also, fa syntax tree is constructed for each node in each CFG.

During the construction of G and G', a syntax tree is also constructed for each node. Though not mentioned earlier, syntax trees for G can be constructed during the first phase. Each syntax tree represents the structure of the corresponding basic block denoted by a node in the CFG. This construction is carried out for the CFGs of each function in P and P'.

The syntax trees for the *Start* and *End* nodes consist of exactly one node each labeled *Start* and *End*, respectively. The syntax trees for other nodes are constructed using traditional techniques often used by compiler writers. In a syntax tree, a function call is represented by parameter nodes, one for each parameter, and a call node. The call node points to a leaf labeled with the name of the function to be called.

Example 9.7 Syntax trees for some nodes in Figure 9.5 are shown in Figure 9.6. Note the semicolon that labels the tree for node 1 in function `main`. It indicates left to right sequencing of statements or expressions.

Figure 9.6 Syntax trees for some nodes in the CFGs of functions `main`, `g1`, and `g2` of P shown in Figure 9.5. A semicolon (;) indicates left to right sequencing of two or more statements within a node.

A test t is selected only if the execution traces in P and P' differ, i.e., the traces traverse nodes n1 and n2 in the two CFGs that have different syntax trees.

Compare CFGs and select tests: In this step the CFGs for P and P' are compared and a subset of T selected. The comparison starts from the *Start* node of the `main` functions of P and P' and proceeds further down recursively identifying corresponding nodes that differ in P and P'. Only tests that traverse such nodes are selected.

While traversing the CFGs, two nodes $n \in N$ and $n' \in N'$ are considered equivalent if the corresponding syntax trees are identical. Two syntax trees are considered identical when their roots have the same labels and the same corresponding descendants (see Exercise 9.4). Function calls can be tricky. For example, if a leaf node is labeled as a function name *foo*, then the CFGs of the corresponding functions in P and P' must be compared to check for the equivalence of the syntax trees.

Example 9.8 Suppose that the basic blocks in nodes n and n' in G and G' have identical syntax trees shown in Figure 9.7(a). However, these two nodes are not considered equivalent because the CFGs of function `foo` in P, and `foo'` in P', are not identical. The difference in the two CFGs is due to different labels of edges going out of node 1 in the CFGs. In Figure 9.7(b), the edge labeled f goes to node 3 while the edge with the same label goes to node 2 in Figure 9.7(c).

`Procedure for selecting modification traversing tests`

Input:
- G, G': CFGs of programs P and P' and syntax trees corresponding to each node in a CFG.
- Test vector $test(n)$ for each node n in each CFG.
- T: Set of non-obsolete tests for P.

Output: T': A subset of T.

Figure 9.7 Syntax trees with identical function call nodes but different function CFGs. (a) Syntax tree for nodes n and n', respectively, in G and G'. (b) CFG for `foo` in P. (c) CFG for `foo'` in P'.

Procedure: `SelectTestsMain`
/*
After initialization, procedure `SelectTests` is invoked. In turn `SelectTests` recursively traverses G and G' starting from their corresponding start nodes. A node n in G found to differ from its corresponding node in G' leads to the selection of all tests in T that traverse n.
*/

Step 1 Set $T' = \emptyset$. Unmark all nodes in G and in its child CFGs.
Step 2 Call procedure `SelectTests`(G.Start, G'.Start'), where G.Start and G'.Start are, respectively, the start nodes in G and G'.
Step 3 T' is the desired test set for regression testing P'.

`End of Procedure SelectTestsMain`
Procedure: `SelectTests` (N, N')
Input: N, N', where N is a node in G and N' its corresponding node in G'.
Output: T'.

Step 1 Mark node N to ensure that it is ignored in the next encounter.
Step 2 If N and N' are not equivalent, then $T' = T' \cup test(N)$ and return, otherwise go to the next step.
Step 3 Let S be the set of successor nodes of N. Note that S is empty if N is the *End* node.
Step 4 Repeat the next step for each $n \in S$.

4.1 If n is marked then return else repeat the following steps.

4.1.1 Let $l = label(N, n)$. The value of l could be `t`, `f`, or ϵ (for empty).
4.1.2 $n' = getNode(l, N')$. n' is the node in G' that corresponds to n in G. Also, the label on edge (N', n') is l.
4.1.3 `SelectTests`(n, n').

Step 5 Return from `SelectTests`.

`End of Procedure SelectTests`

Example 9.9 Next we illustrate the regression test selection procedure using Program P9.1. The CFGs of functions `main`, `g1`, and `g2` are shown in Figure 9.5. *test*(n) for each node in the three functions is given in Example 9.6.

Now suppose that function `g1` is modified by changing the condition at line 3 as shown in Program P9.2. The CFG of `g1` changes

only in that the syntax tree of the contents of node 1 is now different from that shown in Figure 9.5. We will use the `SelectTests` procedure to select regression tests for the modified program. Note that all tests in T in Example 9.5 are valid for the modified program and hence are candidates for selection.

Program P9.2

```
1   int g1(int a, b){        ← Modified g1.
2   int a, b;
3   if(a−1==b)        ← Predicate modified.
4    return(a*a);
5   else
6    return(b*b);
7   }
```

Let us follow the steps as described in `SelectTestsMain`. G and G' refer to, respectively, the CFGs of Program P9.1 and its modified version–the only change being in `g1`.

`SelectTestsMain`.Step 1: $T' = \emptyset$.

`SelectTestsMain`.Step 2: `SelectTests(G.main.Start, G'.main.Start)`.

`SelectTests`.Step 1: $N =$ G.`main`.Start and $N' =$ G'.`main`.Start. Mark G.`main`.Start.

`SelectTests`.Step 2 G.`main`.Start and G'.`main`.Start are equivalent hence proceed to the next step.

`SelectTests`.Step 3 $S = succ$(G.Start) $=$ {G.`main`.1}.

`SelectTests`.Step 4: Let $n =$ G.`main`.1.

`SelectTests`.Step 4.1: n is unmarked, hence proceed further.

`SelectTests`.Step 4.1.1: $l = label$(G.`main`.Start, n) $= \epsilon$.

`SelectTests`.Step 4.1.2: $n' = getNode(\epsilon$, G'.main.Start) $=$ G'.`main`.1.

`SelectTests`.Step 4.1.3: `SelectTests`(n, n').

`SelectTests`.Step 1: $N =$ G.`main`.1 and $N' =$ G'.`main`.1. Mark G.`main`.1.

`SelectTests`.Step 2 G.`main`.1 and G'.`main`.1 are equivalent hence proceed to the next step.

`SelectTests`.Step 3 $S = succ$(G.`main`.1) $=$ {G.`main`.2, G.`main`.3}.

`SelectTests`.Step 4: Let $n =$ G.`main`.2.

`SelectTests`.Step 4.1: n is unmarked hence proceed further.

`SelectTests`.Step 4.1.1: $l = label$(G.`main`.1, n) = `t`.

`SelectTests`.Step 4.1.2: $n' = getNode(l$, G'.`main`.1) = G'.`main`.2.

`SelectTests`.Step 4.1.3: `SelectTests`(n, n').

`SelectTests`.Step 1: $N =$ G.`main`.2 and $N' =$ G'.`main`.2. Mark G.`main`.2.

As G.`main`.2 contains a call to `g1`, the equivalence needs to be checked with respect to the CFGs of `g1` and `g2`. N and N' are not equivalent due to the modification in `g1`. Hence $T' = tests(N) = tests$(G.`main`.2) $= \{t_1, t_3\}$. This call to `SelectTests` terminates. We continue with the next element of S. Exercise 9.6 asks you to complete the steps in this example.

9.5.3 Handling function calls

The `SelectTests` algorithm compares respective syntax trees while checking for the equivalence of two nodes. In the event the nodes being checked contain a call to function f that has been modified to f', a simple check as in Example 9.9 indicates non-equivalence if f and f' differ along any one of the corresponding nodes in their respective CFGs. This might lead to selection of test cases that do not execute the code corresponding to a change in f.

Finding the equivalence of two corresponding nodes in the CFGs of P and P' requires careful attention to any function calls within the nodes.

Example 9.10 Suppose that `g1` in Program P9.1 is changed by replacing line 4 by `return` (a*a*a). This corresponds to a change in node 2 in the CFG for `g1` in Figure 9.5. It is easy to conclude that despite t_1 not traversing node 2, `SelectTests` will include t_1 in T'. Exercise 9.7 asks you to modify `SelectTests`, and the algorithm to check for node equivalence, to ensure that only tests that touch the modified nodes inside a function are included in T'.

9.5.4 Handling changes in declarations

The `SelectTests` algorithm selects modification traversing tests for regression testing. Suppose that a simple change is made to variable declaration and that this declaration occurs in the main function. `SelectTests` will be unable to account for the change made simply because we have not included declarations in the CFG.

A CFG might not capture changes in declarations. A simple way to tackle this problem is to have one node in the CFG for each declaration. Thus, a test will be selected if the corresponding trace traverses nodes containing declarations that differ. However, this method will likely lead to the selection of all tests as each will traverse a declaration node.

One method to account for changes in declarations is to add a node corresponding to the declarations in the CFG of a function. This is done for the CFG of each function in P. Declarations for global variables belong to a node placed immediately following the `Start` node in the CFG for `main`.

The addition of a node representing declarations will force `SelectTests` to compare the corresponding declaration nodes in the CFGs for P and P'. Tests that traverse the declaration node will be included in T' if the nodes are found not equivalent. The problem now is that any change in the declaration will force the inclusion of all tests from T in T'. This is obviously due to the fact that all tests traverse the node following the `Start` node in the CFG for `main`. Next we present another approach to test selection in the presence of changes in declarations.

Let $declChange_f$ be the set of all variables in function f whose declarations have changed in f'. Variables removed or added are not included in $declChange_f$ (see Exercise 9.8). Similarly, we denote by *gdeclChange* the set of all global variables in P whose declarations have changed.

Let $use_f(n)$ be the set of variable names used at node n in the CFG of function f. This set can be computed by traversing the CFG of each function and analyzing the syntax tree associated with each node. Any variable used–not assigned to–in an expression at node n in the CFG of function f is added to $use_f(n)$. Note that $declChange_f$ is empty when there has been no change in declarations of variables in f. Similarly, $use_f(n)$ is empty when node n in CFG(f) does not use any variable, e.g. in statement $x = 0$.

Procedure `SelectTestsMainDecl` is a modified version of procedure `SelectTestsMain`. It accounts for the possibility of changes in declarations and carefully selects only those tests that need to be run again for regression testing.

```
Procedure for selecting modification traversing
tests while accounting for changes in variable
declarations.
```

Input:
- G, G': CFGs of programs P and P' and syntax trees corresponding to each node in a CFG.
- Test vector $test(n)$ for each node n in each CFG.
- For each function f, $use_f(n)$ for each node n in CFG (f).
- For each function f, $declChange_f$.
- The set of global variables whose declarations have changed: *gdeclChange*.
- T: Set of non-obsolete tests for P.

Output: T': A subset of T.

Procedure: `SelectTestsMainDecl`
/*

Procedure `SelectTestsDecl` is invoked repeatedly after the initialization step. In turn `SelectTestsDecl` looks for any changes in declarations and selects those tests from T that traverse nodes affected by the changes. Procedure `SelectTests`, as decribed earlier, is called upon the termination of `processDecl`.

*/

Step 1 Set $T' = \emptyset$. Unmark all nodes in G and in its child CFGs.

Step 2 For each function f in G, call procedure `SelectDeclTest` (f, $declChange_f$, $gdeclChange$). Each call updates T'.

Step 3 Call procedure `SelectTest`(G.Start, G'.Start'), where G.Start and G'.Start are, respectively, the start nodes in G and G'. This procedure may add new tests to T'.

Step 4 T' is the desired test set for regression testing P'.

`End of Procedure SelectTestsMainDecl`

Procedure: `SelectTestsDecl` (f, $declChange_f$, $gdeclChange$)

Input: f is function name and $declChange_f$ is a set of variable names in f whose declarations have changed.

Output: T'.

Step 1 Repeat the following step for each node $n \in \text{CFG}(f)$.

1.1 `if` $use(n) \cap declChange_f \neq \emptyset$ or $use(n) \cap gdeclChange \neq \emptyset$ then $T' = T' \cup test(n)$.

`End of Procedure SelectTestsDecl`

The `SelectTests` procedure remains unchanged. Note that in the approach described above, the CFG of each function does not contain any nodes corresponding to declarations in the function. Declarations involving explicit variable initializations are treated in two parts: a pure declaration part followed by an initialization part. Initializations are included in a separate node placed immediately following the `Start` node. Thus any change in the initialization part of a variable declaration, such as "`int` x=0;" changed to "`int` x=1;" is processed by `SelectTests`.

Example 9.11 Consider Program P9.3 and its CFG shown next to the code in Figure 9.8. As indicated, suppose that the type of variable `z` is changed from `int` to `float`. We refer to the original and the modified programs as P and P', respectively. It is easy to see that $gdeclChange = \emptyset$ and $declChange_{main} = \{z\}$. Suppose the following test set is used for testing P.

Program P9.3

```
1   main(){
2   int x, y, z;        ← Replaced by int x, y; float z;
3   z = 0;
4   input (x, y);
5   if (x < y)
6     {z = x + y; output(z);}
7   output((float) (x – y));
8   end
9   }
```

Figure 9.8 Program and its CFG for Example 9.11.

$$T = \left\{ \begin{array}{ll} t_1: & < x = 1, y = 3 > \\ t_2: & < x = 2, y = 1 > \\ t_3: & < x = 3, y = 4 > \end{array} \right\}$$

We can easily trace CFG(P) for each test case to find the test vectors. These are listed below:

$$\begin{array}{ll} test(1): & \{t_1, t_2, t_3\} \\ test(2): & \{t_1, t_3\} \\ test(3): & \{t_1, t_2, t_3\} \end{array}$$

Abbreviating use_{main} as use, Step 1 in Procedure `SelectTestsDecl` proceeds as follows:

node 1: $use(1) \cap declChange_{main} = \emptyset$. Hence T' does not change.
node 2: $use(2) \cap declChange_{main} = \{z\}$. Hence $T' = T' \cup test(2) = \{t_1, t_3\}$.
node 3: $use(3) \cap declChange_{main} = \emptyset$. Hence T' does not change.

Procedure `SelectTestsDecl` terminates at this point. Procedure `SelectTests` does not change T' as all the corresponding nodes in CFG(P) and CFG(P') are equivalent. Hence we obtain the regression test $T' = \{t_1, t_3\}$.

Selection of regression tests using the execution trace method may lead to a larger than necessary test suite.

9.6 Test Selection Using Dynamic Slicing

Selection of modification traversing tests using execution trace may lead to regression tests that are really not needed. Consider the following scenario. Program P has been modified by changing the statement at line l. There

Program P9.4

```
1   main(){
2   int p, q, r, z;
3   z = 0;
4   input (p, q, r);
5   if (p < q)
6     z = 1;        ← This statement is modified to z = –1
7   if (r > 1)
8     z = 2;
9   output (z);
10  end
11  }
```

Figure 9.9 A program and its CFG for Example 9.12.

are two tests t_1 and t_2 used for testing P. Suppose that both traverse l. The execution slice technique described earlier will select both t_1 and t_2.

Now suppose that whereas t_1 traverses l, the statement at l does not affect the output of P along the path traversed from the `Start` to the `End` node in CFG(P). On the contrary, traversal of l by t_2 does affect the output of P. In this case there is no need to test P' on t_1.

Example 9.12 Consider Program P9.4 that takes three inputs, computes, and outputs z. Suppose that P is modified by changing the statement at line 6 as shown.
Consider the following test set used for testing P:

$$T = \left\{ \begin{array}{ll} t_1: & <p=1, q=3, r=2> \\ t_2: & <p=3, q=1, r=0> \\ t_3: & <p=1, q=3, r=0> \end{array} \right\}$$

Tests t_1 and t_3 traverse node 2 in CFG(P) shown in Figure 9.9, t_2 does not. Hence if we were to use `SelectTests` described earlier, then $T' = \{t_1, t_3\}$. This T' is a set of modification traversing tests. However, it is easy to check that even though t_1 traverses node 2, output z does not depend on the value of z computed at this node. Hence, there is no need to test P' against t_1; it needs to be tested only against t_3.

We will now present a technique for test selection that makes use of program dependence graph and dynamic slicing to select regression tests. The main advantage of this technique over the technique based exclusively on

the execution slice is that it selects only those tests that are modification traversing and might affect the program output.

9.6.1 Dynamic slicing

Given a test t variable v in program *P*, and an execution trace of *P* with respect to t, a dynamic slice is the set of statements in the execution trace that did affect v.

Let *P* be the program under test and *t* a test case against which *P* has been executed. Let *l* be a location in *P* where variable *v* is used. The dynamic slice of *P* with respect to *t* and *v* is the set of statements in *P* that lie in *trace*(*t*) and did effect the value of *v* at *l*. Obviously, the dynamic slice is empty if location *l* was not traversed during this execution. The notion of a dynamic slice grew out of a static slice based on program *P* and not on its execution.

Example 9.13 Consider *P* to be Program P9.4 in Figure 9.9 and test t: $< p = 1, q = 3, r = 2 >$. The dynamic slice of *P* with respect to variable *z* at line 9 and test t_1 consists of statements at lines 4, 5, 7, and 8. The static slice of *z* at line 9 consists of statements at lines 3, 4, 5, 6, 7, and 8. It is generally the case that a dynamic slice for a variable is smaller than the corresponding static slice. For t: $< p = 1, q = 0, r = 0 >$, the dynamic slice contains 3, 4, 5, and 7. The static slice does not change.

9.6.2 Computation of dynamic slices

A precise dynamic slice consists of exactly the statements that affect a given variable v at a given point in the program. Not all algorithms for computing a dynamic slice will always give a precise dynamic slice.

The dynamic dependence graph is used in computing the dynamic slice of *P* with respect to a variable v at a given location and test t.

There are several algorithms for computing a dynamic slice. These vary across attributes such as the precision of the computed slice and the amount of computation and memory needed. A precise dynamic slice is one that contains exactly those statements that might affect the value of variable *v* at the given location and for a given test input. We use $DS(t, v, l)$ to denote the dynamic slice of variable *v* at location *l* with respect to test *t*. When the variable and its location are easily identified from the context, we simply write *DS* to refer to the dynamic slice.

Next we present an algorithm for computing dynamic slices based on the dynamic dependence graph. Several other algorithms are cited in the bibliography section. GIven program *P*, test *t*, variable *v*, and location *l*, computation of a dynamic slice proceeds in the following steps.

Procedure: `DSLICE`

Step 1 Execute *P* against test case *t* and obtain *trace*(*t*).
Step 2 Construct the dynamic dependence graph G from *P* and *trace*(*t*).
Step 3 Identify in G the node *n* labeled *l* and containing the last assignment to *v*. If no such node exists then the dynamic slice is empty, otherwise proceed to the next step.

Step 4 Find in G the set $DS(t, v, n)$ of all nodes reachable from n, including n. $DS(t, v, n)$ is the dynamic slice of P with respect to variable v at location l for test t.

```
End of Procedure DSLICE
```

In Step 2 of `DSLICE` we construct the dynamic dependence graph (DDG). This graph is similar to the program dependence graph (PDG) introduced in Chapter 1. Given program P, a PDG is constructed from P whereas a DDG is constructed from the execution trace $trace(t)$ of P. Thus, statements in P that are not in $trace(t)$ do not appear in the DDG.

While a program dependence graph is constructed from the given program, the dynamic program dependence graph is constructed from a trace of P for a given test case.

Construction of G begins by initializing it with one node for each declaration. These nodes are independent of each other and no edge exists between them. Then a node corresponding to the first statement in $trace(t)$ is added. This node is labeled with the line number of the executed statement. Subsequent statements in $trace(t)$ are processed one by one. For each statement, a new node n is added to G. Control and data dependence edges from n are then added to the existing nodes in G. The next example illustrates the process.

Example 9.14 Consider the program in Figure 9.10. We have ignored the function header and declarations as these do not affect the computation of the dynamic slice in this example (see Exercise 9.11). Suppose now that 9.14 is executed against test case $t :< x = 2, y = 4 >$. Also, we assume that successive values of x are 0 and 5. Function $f1(x)$ evaluates to 1, 2, and 3, respectively, for $x = 2, 0,$ and 5. Under these assumptions we get $trace(t) = (1, 2^1, 3^1, 4, 6^1, 7^1, 2^2, 3^2, 5, 6^2, 7^2, 8)$; superscripts differentiate multiple occurrences of a node.

The construction of DDG is exhibited in Figure 9.10 above the solid line. To begin with, node labeled 1 is added to G. Node 2 is added next. This node is data independent on node 1 and hence an edge, indicated as a solid line, is added from node 2 to node 1.

Next, node 3 is added. This node is data dependent on node 1 as it uses variable x defined at node 1. Hence an edge is added from node 3 to node 1. Node 3 is also control dependent on node 2 and hence an edge, indicated as a dotted line, is added from node 3 to node 2. Node 4 is added next and data and control dependence edges are added, respectively, from node 4 to node 1 and to node 3. The process continues as described until the node corresponding to the last statement in the trace, i.e. at line 8, is added. The final DDG is shown in Figure 9.10 below the solid line.

Program P9.5

```
1   input (x, y);
2   while (x < y){
3     if (f1(x)==0)
4       z = f2(x);
      else
5       z = f3(x);
6     x = f4(x);
7     w = f5(z);
    }
8   output (w);
    end
```

Figure 9.10 A program and its dynamic dependence graph for Example 9.14 and test $< x = 2, y = 4 >$. Function header and declarations omitted for simplicity. Construction process for the first four nodes is shown above the solid line.

The dynamic program dependence graph is used to obtain dynamic slices.

The DDG is used for the construction of a dynamic slice as mentioned in Steps 3 and 4 of procedure `DSLICE`. When computing the dynamic slice for the purpose of regression test selection, this slice is often based on one or more output variables which is w in Program P9.5.

Example 9.15 To compute the dynamic slice for variable w at line 8, we identify the last definition of w in the DDG. This occurs at line 7 and, as marked in the figure, corresponds to the second occurrence of node 7 in Figure 9.10. We now trace backwards starting at node 7 and collect all reachable nodes to obtain the desired dynamic slice as {1, 2, 3, 5, 6, 7, 8} (also see Exercise 9.10).

9.6.3 Selecting tests

Dynamic slices are computed for all or a selected subset of the output variables of a program.

Given a test set T for P, we compute the dynamic slice of all, or a selected set of, output variables for each test case in T. Let $DS(t)$ denote such a dynamic slice for $t \in T$. $DS(t)$ is computed using the procedure described in the previous section. Let n be a node in P modified to generate P'. Test $t \in T$ is added to T' if n belongs to $DS(t)$.

We can use `SelectTests` to select regression tests but with a slightly modified interpretation of $test(n)$. We now assume that $test(n)$ is the set of

tests $t \in T$ such that $n \in DS(t)$. Thus, for each node n in CFG(P), only those tests that traverse n and might have effected the value of at least one of the selected output variables are selected and added to T'.

Example 9.16 Suppose that line 4 in Program P9.5 is changed to obtain P'. Should t from Example 9.14 be included in T'? If we were to use only the execution slice, then t will be included in T' because it causes the traversal of node 4 in Figure 9.10. However, the traversal of node 4 does not effect the computation of w at node 8 and hence, when using the dynamic slice approach, t is not included in T'. Note that node 4 is not in $DS(t)$ with respect to variable w at line 8 and hence t should not be included in T'.

9.6.4 Potential dependence

The dynamic slice contains all statements in *trace*(t) that had an effect on program output. However, there might be a statement s in the trace that did not effect program output but may affect if changed. By not including s in the dynamic slice we exclude t from the regression tests. This implies that an error in the modified program due to a change in s might go detected.

A dynamic slice contains all statements in a program that had an impact on an output variable.

Example 9.17 Let P denote Program P9.6. Suppose P is executed against test case t: $< N = 1, x = 1 >$ and that $f(x) < 0$ during the first and only iteration of the loop in P. We obtain $trace(t) = (1, 2, 3, 4, 5, 6, 8, 10, 4, 11)$. The DDG for this trace is shown in Figure 9.11; for this example ignore the edges marked as "p." The dynamic slice $DS(t, z, 11) = \{3, 11\}$ for output z at location 11.

Program P9.6

```
     int  x, z, i, N;
1    input (N);
2    i=1;
3    z=0;
4    while (i< N){
5     input (x);
6     if (f(x)==0)        ← Erroneous condition.
7      z=1;
8     if (f(x)>0)
9      z=2;
10    i++;
     }
11   output (z);
     end
```

Notice that $DS(t, z, 11)$ does not include the node corresponding to line 6. Hence t will not be selected for regression testing of P' obtained by modifying the `if` statement at line 6. But, of course, t must be included in the regression test.

Potential dependance is used to include test cases that would otherwise be excluded when using the dynamic slicing algorithm mentioned earlier.

We define the concept of *potential dependence* to overcome the problem illustrated in the previous example. Let *trace*(t) be a trace of P on test t. Let v be a variable used at location Lv and p a predicate that appears at location Lp prior to the traversal of location Lv. A potential dependency is said to exist between v and p when the following two conditions hold:

1. v is never defined on the subpath traversed from Lp to Lv but there exists another path, say r, from Lp to Lv where v is defined.
2. Changing the evaluation of p may cause path r to be executed.

The following example illustrates how to use the above definition to identify potential dependencies.

Example 9.18 Let us apply the above definition of potential dependence to Program P9.6 when executed against t as in Example 9.17. The subpath traversed from node 6 to node 11 contains nodes in the following sequence: 6, 8, 10, 4, 11. Node 11 has a potential dependency on node 6 because (i) z is never defined on this subpath but there exists a path r from node 6 to node 11 along which z is defined and (ii) changing the evaluation of the predicate at node 6 causes r to be executed. This potential dependency is shown in Figure 9.11. Note that one possible subpath r contains nodes in the following sequence: 6, 7, 8, 10, 4, 11.

Figure 9.11 Dynamic dependence graph for Program P9.6 obtained from *trace*(t), t: $< N = 1, x = 1 >$. Unmarked dotted edges indicate control dependence. Edges marked "p" indicate potential dependence.

Using a similar argument it is easy to show the existence of potential dependency between nodes 8 and 11. The two potential dependencies are shown by dotted edges labeled "*p*" in Figure 9.11.

`Procedure to compute potential dependencies.`

Input:
- Program *P* and its CFG G. There is exactly one node in G for each statement in *P*.
- *trace*(*t*) for test *t* obtained by executing *P* against *t*.
- DDG(*t*).
- Location L and variable *v* in *P* for which the potential dependencies are to be computed.

Output: PD: Set of edges indicating potential dependencies between *L* and other nodes in DDG(*t*).

Procedure: `ComputePotentialDep`

/*

Procedure `ComputePotentialDep` is invoked after the construction of the dynamic dependence graph G corresponding to a given trace. It uses G to determine reaching definitions of *v* at location L. These definitions are then used to iteratively compute PD.

*/

Step 1 Using *P* and G compute the set *S* of all nodes in G that contain (static) reaching definitions of *v* at location L. A *reaching definition* of *v* at node L is a node Lv in G that assigns a value to *v* and there is a path from the start node in G to the end node that reaches Lv and then L without redefining *v*. We refer to these as *static* reaching definitions because they are computed from *P* and not from the execution trace of *P*.

For each program variable *v* defined at location L, there exist a set of nodes known as the "reaching definition." A node Lv is in this set if there is a path from the start of the CFG to Lv and then to node L without an intermediate redefinition of v.

Static reaching definitions are computed from *P* and not from its execution trace.

Step 2 Compute the set *C* of direct and indirect control nodes. A direct control node *n* is a predicate node in G such that some node $m \in S$ has control dependency on *n*. All nodes that are control dependent on nodes in *C* are indirect control nodes and also added to C.

Step 3 Find node *D* in G that contains the last definition of *v* before control arrives at L in *trace*(*t*). In the absence of such a node, *D* is the declaration node corresponding to *v*.

Step 4 Let PD = Ø.

Step 5 Let *nodeSeq* be the sequence of nodes in *trace*(*t*) contained between the occurrences of Lv and L, and including Lv and L. Mark each node in *nodeSeq* as "NV".

Step 6 Repeat the following steps for each *n* in *nodeSeq* starting with $n = L$ and moving backwards.

6.1 Execute the following steps if n is marked "NV."

6.1.1 Mark n as "V."

6.1.2 If $n \in C$ then do the following.

(a) PD = PD$\cup\{n\}$.

(b) Let M be the set of nodes in *nodeSeq* between n and D. For each node $n' \in M$, mark as "V" all nodes $m \in G$ such that n' is control dependent on m.

```
End of Procedure ComputePotentialDep
```

Example 9.19 Let us apply procedure `ComputePotentialDep` to compute potential dependencies for variable z at location 11 for $trace(t) = (1, 2, 3, 4, 5, 6, 8, 10, 4^2, 11)$ as in Example 9.17. The inputs to `ComputePotentialDep` include data from Program P9.6, G shown in Figure 9.11, $trace(t)$, location L = 11, and variable $v = z$.

Step 1: From P we get the static reaching definitions of v at L as $S = \{3, 7, 9\}$. Note that in general this is a difficult task for large programs and especially those that contain pointer references (see bibliographic notes for general algorithms for computing static reaching definitions).

Step 2: Node 3 has no control dependency. Noting from P, nodes 7 and 9 are control dependent on, respectively, nodes 6 and 8. Each of these nodes is in turn control dependent on node 4. Thus we get $C = \{4, 6, 8\}$.

Step 3: Node 3 contains the last definition of z in $trace(t)$. Hence $D = z$.

Step 4: PD = Ø.

Step 5: $Lv = 3$. $nodeSeq = (3, 4, 5, 6, 8, 10, 4^2, 11)$. Each node in *nodeSeq* is marked NV. We show this as *nodeSeq* = (3^{NV},4^{NV}, 5^{NV} 6^{NV}, 8^{NV}, 10^{NV}, 4^{NV}, 11^{NV}).

Step 6: Select n = L = 11.

Step 6.1: n is marked NV hence we consider it.

Step 6.1.1: *nodeSeq* = (3^{NV},4^{NV}, 5^{NV} 6^{NV}, 8^{NV}, 10^{NV}, 4^{NV}, 11^{V}). We have omitted the superscript 2 on the second occurrence of 4.

Step 6.1.2: As n is not in C, we ignore it and continue with the loop.

Step 6: Select n = L = 4.

Step 6.1: n is marked NV hence we consider it.

Step 6.1.1: *nodeSeq* = (3^{NV},4^{NV}, 5^{NV} 6^{NV}, 8^{NV}, 10^{NV}, 4^{V}, 11^{V}).

Step 6.1.2: As n is in C, we process it. (a) PD = {4}. (b) Nodes 4 and 11 are not control dependent on any node and hence no other node needs to be marked.

Steps 6, 6.1, 6.1.1: Select n = 10. As n is marked NV we consider it and mark it V. *nodeSeq* = (3^{NV},4^{NV}, 5^{NV} 6^{NV}, 8^{NV}, 10^{V}, 4^{V}, 11^{V}).

Step 6.1.2: Node 10 is not a control node hence we ignore it and move to the next iteration.

Steps 6, 6.1, 6.1.1: Select $n = 8$. This node is marked NV, hence we mark it V and process it. *nodeSeq* = (3^{NV},4^{NV}, 5^{NV} 6^{NV}, 8^{V}, 10^{V}, 4^{V}, 11^{V}).

Step 6.1.2: Node 6 is in C hence (a) we add it to PD to get PD = {4, 8} and (b) mark node 4 as V because 8 is control dependent on these nodes. *nodeSeq* = (3^{NV},4^{V}, 5^{NV} 6^{NV}, 8^{V}, 10^{V}, 4^{V}, 11^{V}).

Steps 6, 6.1, 6.1.1: Select $n = 6$. This node is marked NV, hence we mark it V and process it. *nodeSeq* = (3^{NV},4^{NV}, 5^{NV} 6^{V}, 8^{V}, 10^{V}, 4^{V}, 11^{V}).

Step 6.1.2: Node 8 is in C hence (a) we add it to PD to get PD = {4, 6, 8} and (b) there are no new nodes to be marked.

Steps 6, 6.1, 6.1.1, 6.1.2: Select $n = 5$. This node is marked NV, hence we mark it V and process it. *nodeSeq* = (3^{NV},4^{NV}, 5^{V} 6^{V}, 8^{V}, 10^{V}, 4^{V}, 11^{V}). We ignore 5 as it is not in C.

The remaining nodes in *nodeSeq* are either marked V or are not in C and hence will be ignored. The output of procedure `ComputePotentialDep` is PD = {4, 6, 8}.

Computing potential dependencies in the presence of pointers is a complex task. Several algorithms for computing such dependencies are available.

Computing potential dependences in the presence of pointers requires a complicated algorithm. See the bibliography section for references to algorithms for computing dependence relationships in C programs.

9.6.5 Computing the relevant slice

A relevant slice of variable v in program P at location L and for a test input t is the set of all nodes in the program trace which if modified may alter the program output.

The *relevant slice* $RS(t, v, n)$ with respect to variable v at location n for a given test case t is the set of all nodes in the *trace*(t) which if modified may alter the program output. Given *trace*(t), the following steps are used to compute the relevant slice with respect to variable v at location n.

A relevant slice is derived from the dynamic program dependence graph by adding addition edges that correspond to potential dependence.

Step 1 Using DDG, compute the dynamic slice $DS(t, v, n)$ corresponding to node n which contains the output variable v.

Step 2 Modify the DDG by adding edges corresponding to all potential dependencies from node n to a predicate node.

Step 3 Find the set S of all nodes reachable from any node in $DS(t, v, n)$ by following the data and potential dependence (not control dependence) edges.

Step 4 The relevant slice $RS(t, v, n) = S \cup DS(t, v, n)$.

Example 9.20 Continuing with Example 9.19 and referring to Figure 9.11 that indicates the potential dependence, we obtain $S = \{1, 4, 5, 6\}$, and hence $RS(t, z, 11) = S \cup DS(t, z, 11) = \{1, 3, 4, 5, 6, 11\}$. If the relevant slice is used for test selection, test t in Example 9.17 will be selected if any statement on lines 1, 3, 4, 5, 6, and 11 is modified.

9.6.6 Addition and deletion of statements

Modification of a program via the addition on the deletion of statements may affect some relevant slices.

Statement addition: Suppose that program P is modified to P' by adding statement s. Obviously, no relevant slice for P would include the node corresponding to s. However, we do need to find tests in T, used for testing P, that must be included in the regression test set T' for P'. To do so, suppose that (a) s defines variable x, as for example through an assignment statement, and (b) $RS(t, x, l)$ is the relevant slice for variable x at location l corresponding to test case $t \in T$. We use the following steps to determine whether or not t needs to be included in T'.

Step 1 Find the set S of statements $s_1, s_2, \ldots, s_k, k \geq 0$, that use x. Let $n_i, 1 \leq i \leq k$ denote the node corresponding to statement s_i in the DDG of P. A statement in S could be an assignment that uses x in an expression, an output statement, or a function or method call. Note that x could also be a global variable in which case all statements in P that refer to x must be included in S. It is obvious that if $k = 0$ then the newly added statement s is useless and need not be tested.

Step 2 For each test $t \in T$, add t to T' if for any $j, 1 \leq j \leq k, n_j \in RS(t, x, l)$.

Example 9.21 Suppose that Program P9.5 is modified by adding the statement $x = g(w)$ immediately following line 4 as part of the `then` clause. The newly added statement will be executed when control arrives at line 3 and $f1(x) = 0$.

Set S of statements that use x is $\{2, 3, 4, 5, 6\}$. Now consider test t from Example 9.14. You may verify that the dynamic slice of t with respect to variable w at line 8 is the same as its relevant slice $RS(8)$ which is $\{1, 2, 3, 5, 6, 7, 8\}$ as computed in Example 9.15. Several

statements in S belong to $RS(t, w, 8)$ and hence t must be included in T' (also see Exercise 9.17).

Statement deletion: Now suppose that P' is obtained by deleting statement s from P. Let n be the node corresponding to s in the DDG of P. It is easy to see that all tests from T whose relevant slice includes n must be included in T'.

Statement deletion and addition: An interesting case arises when P' is obtained by replacing some statement s in P by s' such that the left side of s' is different from that of s. Suppose that node n in the DDG of P corresponds to s and that s' modifies variable x. This case can be handled by assuming that s has been deleted and s' added. Hence all tests $t \in T$ satisfying the following conditions must be included in T' (a) $n \in RS(t, w, l)$ and (b) $m \in RS(t, w, l)$ where node m in CFG(P) corresponds to some statement in P that uses variable w.

P' can be obtained by making several other types of changes in P not discussed above. See Exercise 9.18 and work out how the relevant slice technique can be applied for modifications other than those discussed above.

9.6.7 Identifying variables for slicing

You might have noticed that we compute a relevant slice with respect to a variable at a location. In all examples so far, we used a variable that is part of an output statement. However, a dynamic slice can be constructed based on any program variable that is used at some location in P, the program that is being modified. For example, in Program 9.17 one may compute the dynamic slice with respect to variable z at line 9 or at line 10.

In large programs, the identification of variables and their locations could become an enormous task. Thus, one needs to carefully identify only the most critical variables and the locations where they are defined.

Some programs will have several locations and variables of interest at which to compute the dynamic slice. One might identify all such locations and variables, compute the slice of a variable at the corresponding location, and then take the union of all slices to create a combined dynamic slice (see Exercise 9.10). This approach is useful for regression testing of relatively small components.

In large programs there might be many locations that potentially qualify to serve as program outputs. The identification of all such locations might itself be an enormous task. In such situations a tester needs to identify critical locations that contain one or more variables of interest. Dynamic slices can then be built only on these variables. For example, in an access control application found in secure systems, such a location might be immediately following the code for processing an activation request. The state variable might be the variable of interest.

9.6.8 Reduced dynamic dependence graph

The size of a dynamic dependence graph is unbounded as it is directly proportional to that of the execution trace.

As described earlier, a dynamic dependence graph constructed from an execution trace has one node corresponding to each program statement in the trace. As the size of an execution trace is unbounded, so is that of the DDG. Here we describe another technique to construct a reduced dynamic dependence graph (RDDG). While an RDDG looses some information that exists in a DDG, the information loss does not impact the tests selected for regression testing (see Exercise 9.25). Furthermore, the technique for the construction of an RDDG does not require saving the complete trace in memory.

A reduced dynamic dependence graph contains at most one occurrence of each node in the program trace.

Construction of an RDDG, G proceeds during the execution of P against a test t. For each executed statement s at location l, a new node n labeled l is added to G only when there does not already exist such a node. In case n is added to G, any of its control and data dependencies are also added. In case n is not added to G, the control and data dependences of n are updated. The number of nodes in the RDDG so constructed equals at most the number of distinct locations in P. In practice, however, most tests exercise only a portion of P thus leading to a much smaller RDDG.

Example 9.22 Suppose 9.17 is executed against t and $trace(t) = \{1, 2, 3, 4^1, 5, 6, 8, 9, 10, | 4^2, 5^2, 6^2, 8^2, 10^2, | 4^3, 5^3, 6^3, 7, 8^3, 10^3, | 4^4, 11\}$. Construction of RDDG G is shown in Figure 9.12. The vertical bars indicate the end of the first three iterations of the loop. Figure 9.12(a) shows the partial RDDG at the end of the first iteration of the loop, Figure 9.12(b) at the end of the second iteration, and Figure 9.12(c) the complete RDDG at the end of program execution. We have ignored the declaration node in this example.

Notice that none of the statements in $trace(t)$ corresponds to more than one node in G. Also, note how new nodes and new dependence edges are added. For example, in the second iteration, a data dependence edge is added from node 4 to node 10, and from 10 to itself. The RDDG contains only 11 nodes in contrast to a DDG which would contain 22 nodes when constructed using the procedure described in Section 9.6.2.

The procedure for obtaining a dynamic slice from an RDDG remains the same as described in Section 9.6.2. To obtain the relevant slice one first discovers the potential dependences and then computes the relevant slice as illustrated in Section 9.6.5.

Figure 9.12 Reduced dynamic dependence graph for Program 9.17 obtained from $trace(t) = \{1, 2, 3, 4^1, 5, 6, 8, 9, 10, | \ 4^2, 5^2, 6^2, 8^2, 10^2, | \ 4^3, 5^3, 6^3, 7, 8^3, 10^3, | \ 4^4, 11\}$. Intermediate status starting from (a) node 1 until node 10, (b) node 4^2 until node 10^2, and (c) complete RDDG for $trace(t)$.

9.7 Scalability of Test Selection Algorithms

The execution slice and dynamic slicing techniques described above have several associated costs. First there is the cost of doing a complete static analysis of program P that is modified to generate P'. While there exist several algorithms for static analysis to discover data and control dependencies, they are not always precise especially for languages that offer pointers. Second, there is the run time overhead associated with the instrumentation required for the generation of an execution trace. While this overhead might be tolerable for operating systems and non-embedded applications, it might not be for embedded applications.

The cost of dynamic slicing includes that of static program analysis and the construction of the dynamic dependence graph for every test input.

In dynamic slicing, there is the additional cost of constructing and saving the dynamic dependence graph for every test. Whether or not this cost can be tolerated depends on the size of the test suite and the program. For programs that run into several hundred thousands of lines of code and are tested using a test suite containing hundreds or thousands of tests, DDG (or the RDDG) at the level of data and control dependence at the statement level might not be cost effective at the level of system, or even integration, testing.

Thus, while both the execution trace and DDG-or RDDG-based techniques are likely to be cost effective when applied to regression testing of components of large systems, they might not be when testing the complete

For very large programs, one might use coarse-level data and control dependence analysis to reduce the cost of generating dynamic dependence graphs.

system. In such cases, one can use a coarse level data and control dependence analysis to generate dependence graphs. One may also use coarse level instrumentation for saving an execution trace. For example, instead of tracing each program statement, one could trace only the function calls. Also, dependence analysis could be done across functions and not across individual statements.

Example 9.23 Consider Program P9.1 in Example 9.5. In that example we generate three execution traces using three tests. The traces are at the statement level.

Suppose that we want to collect the execution trace at the level of function calls. In this case the program needs to be instrumented only at call points, or only at the start of each function definition. Thus the total number of probes added to the program to obtain a function-level execution trace is equal to the number of functions in the program, three in this case–`main`, `g1`, and `g2`. The function traces corresponding to the three tests are shown in the following table:

Test (t)	Execution trace ($trace(t)$)
t_1	main, g1, main
t_2	main, g2, main
t_3	main, g1, main

Notice the savings in the number of entries in each execution trace. The function trace has a total of nine entries for all three tests whereas the statement level trace has 30. In general, the savings in storage requirements for a functional trace over statement trace will depend on the average size of the functions (see Exercise 9.35).

While tracing at a coarse level leads to reductions in the run time overhead, it also leads to a much smaller DDG. For example, rather than represent control and data dependencies across program variables, one can do the same across functions. In most practical applications, data and control dependencies at a coarse level will lead to smaller DDGs relative to those constructed using dependencies at the level of program variables.

In most applications, data and control dependence at a coarse-level will likely lead to smaller dynamic dependence graphs.

Consider two functions f_1 and f_2 in program P. Function f_2 has data dependency on f_1 if there is at least one variable defined in f_1 such that its definition reaches f_2. Similarly, f_2 is control dependent on f_1 when there are at least two paths from the start of P to the end that pass through f_1 but only one of these passes through f_2 after having passed through f_1 and the

Program P9.7

```
1  main(){
2  int x, y, z;
3  input(x, y);
4  z = f1(x);
5  if(z > 0)
6    z = f2(x);
7  output (z);
8  end
9  }
```

```
1  intf1(intx){
2  int p;
3  if(x > 0)
4    p = f3(x, y);
5  return(p);
6  }
```

Figure 9.13 Dynamic data dependence graph for Program P9.7 corresponding to the function trace: `main`, `f1`, `f3`, `f1`, and `main`.

other path might be taken if a condition in f_1 evaluated differently. It is important to note that construction of DDG (or RDDG) requires static analysis of P to determine data and control dependence across various functions.

Example 9.24 Program P9.7 consists of `main` and three functions `f1`, `f2`, and `f3`. Consider the function trace consisting of the following calling sequence: `main`, `f1`, `f3`, `f1`, and `main`. The DDG corresponding to this trace appears in Figure 9.13.

Notice the data and control dependence among various functions in the trace. `main` has data dependence on `f1` due to its use of `f1` to compute the value of z. Function `f1` also has data dependence on `main` due to its use of x. `f3` uses global variable y and hence has data dependence on `main`. Also, `f3` is control dependent on `f1` as it may or may not be called upon entry to `f1` (see Exercise 9.22).

Computing dynamic slices using a DDG based on function traces can be tricky. First, one needs to ask the proper question. In the program variable based slicing discussed earlier, we are able to specify a variable and its location for slicing. Against what should the slice be constructed when traces are based on functions? Second, the DDG exhibits dependencies at the function level, and not at the level of program variables. Hence, how do we relate the slicing requirement to the DDG?

Computation of dynamic slices at the level of functions requires one to specify requirements for slices. This can be done by identifying variables of interest inside functions.

We can compute a dynamic slice based on a program variable at a specific location just as we did earlier. However, one needs to use static analysis to locate the function that contains the variable and its location. For example, if the slice is to be computed for z at line 7 in Program P9.7,

then we know that this variable and location is inside `main`. Once the containing function has been identified, the dynamic slice is constructed using the DDG as before (see Exercise 9.23).

9.8 Test Minimization

Test minimization is performed using one of more coverage criteria. These criteria could be black- or white-box.

Modification traversing test selection finds a subset T' of T to use for regression testing. Now suppose that P contains n testable entities. Functions, basic block, conditions, and definition-use pairs are all examples of testable entities. Suppose that tests in T' cover, i.e. test, $m < n$ of the testable entities. It is likely that there is an overlap among the entities covered by two tests in T'.

We therefore ask: Is it possible, and beneficial, to reduce T' to T'', such that $T'' \subseteq T'$ and each of the m entities covered by tests in T' is also covered by tests in T''? Such a test reduction process is commonly referred in software testing as *test minimization*. Of course, we can also apply test minimization directly to T (see Exercise 9.28).

Example 9.25 Consider Program P9.7 and test set $T = \{t_1, t_2, t_3\}$. Figure 9.14 shows the CFG of P9.7. Let basic block be the entity of interest. The coverage of basic blocks for tests in T is as follows; note that we have excluded functions `f2` and `f3` from the coverage information assuming that these are library functions and that we do not have source code available to measure the corresponding block coverage.

t_1:	`main`:1,2,3	`f1`:1,3
t_2:	`main`:1,3	`f1`:1,3
t_3:	`main`:1,3	`f1`:1,2,3

Tests in T cover all six blocks, three in `main` and three in `f1`. However, it is easy to check that these six blocks can also be covered by t_1 and t_3. Hence rather than using T as a regression test one could also use $T' = \{t_1, t_3\}$, albeit with caution. Note that in this example the minimized test suite is unique.

A reduced (minimized) test suite might not have the same fault detection effectiveness as that of its container test set; however, the reduction in the effectiveness depends on the coverage criteria used. The stronger the coverage criterion, the less the reduction in the effectiveness.

One could obtain a significant reduction in the size of a test suite by applying coverage-based minimization as in the previous example. However, one might ask: *Will the minimized test suite have the same fault detection effectiveness as the corresponding non-minimized version?* The answer to this question depends on the modifications made to P to obtain P', the kind of faults in P', and the entity used for minimization (see Exercise 9.29).

Figure 9.14 Control flow graph of Program P9.7.

9.8.1 The set cover problem

The test minimization problem can formulated as a set cover problem.

The test minimization problem can be formulated as the set cover problem. Let E be a set of entities and TE a set of subsets of E. A set cover is a collection of sets $C \subseteq TE$ such that the union of all elements of C is TE. The set cover problem, more aptly the set cover optimization problem, is to find a minimal C. Relating this to test minimization, set E could be for example, a set of basic blocks, set TE a collection of sets of basic blocks covered by each test in T. The test minimization problem then is to find a minimal subset of TE that covers all elements of E.

Example 9.26 Let us formulate the test minimization problem as set cover problem with reference to Example 9.25. We get the following sets:

$$\begin{aligned} E &= \{\texttt{main.1}, \texttt{main.2}, \texttt{main.3}, \texttt{f1.1}, \texttt{f1.2}, \texttt{f1.3}\} \\ TE &= \{\{\texttt{main.1}, \texttt{main.2}, \texttt{main.3}, \texttt{f1.1}, \texttt{f1.3}\}, \\ &\quad \{\texttt{main.1}, \texttt{main.3}, \texttt{f1.1}, \texttt{f1.3}\}, \\ &\quad \{\texttt{main.1}, \texttt{main.3}, \texttt{f1.1}, \texttt{f1.2}, \texttt{f1.3}\}\} \end{aligned}$$

The solution to the test minimization problem is

$$\begin{aligned} C &= \{\{\texttt{main.1}, \texttt{main.2}, \texttt{main.3}, \texttt{f1.1}, \texttt{f1.3}\}, \\ &\quad \{\texttt{main.1}, \texttt{main.3}, \texttt{f1.1}, \texttt{f1.2}, \texttt{f1.3}\}\} \end{aligned}$$

The greedy algorithms or finding and minimizing a test set T selects test t that covers the maximal number of elements in P; an element might be a block, a decision, or any other entity depending on the minimization criterion.

9.8.2 A procedure for test minimization

There exist several procedures for solving the set cover optimization problem. Given set TE, the naive algorithm computes the power set, 2TE, and selects from it the minimal covering subset. As the size of TE increases, the naive algorithm quickly becomes impractical (see Exercise 9.30).

The *greedy* algorithm is well known. Given a set of tests T and entities E, it begins by finding a test $t \in T$ that covers the maximum number of

The greedy algorithm might fail to find a minimal test set.

entities in E. t is selected, removed from T, and the entities it covers are removed from E. The procedure now continues with the updated T and E. The greedy algorithm terminates when all entities have been covered. While the greedy algorithm is faster than the naive set enumeration algorithm, it fails to find the minimal set cover in certain cases. Below we present a slight variation of the greedy algorithm, named `CMIMX`.

`Procedure to find minimal covering set.`

Input: An $n \times m$ matrix C. $C(i,j)$ is 1 if test t_i covers entity j, else it is 0. Each column of C contains at least one non-zero entry. Each column corresponds to a distinct entity and each row to a distinct test.

Output: MInimal cover $minCov = \{i_1, i_2, \ldots, i_k\}$ such that for each column in C there is at least one non-zero entry in at least one row with index in $minCov$.

Procedure: `CMIMX`

/* This procedure computes the minimal test cover for entities in C. */

Step 1 Set $minCov = \emptyset$. $yetToCover = m$.

Step 2 Unmark each of the n tests and m entities. An unmarked test is still under consideration, while a marked test is one already added to $minCov$. An unmarked entity is not covered by any test in $minCov$, whereas a marked entity is covered by at least one test in $minCov$.

Step 3 Repeat the following steps while $yetToCover > 0$.

3.1 Among the unmarked entities (columns) in C find those containing the least number of 1's. Let LC be the set of indices of all such columns. Note that LC will be non-empty as every column in C contains at least one non-zero entry.

3.2 Among all the unmarked tests (rows) in C that also cover entities in LC, find those that have the maximum number of non-zero entries. Let s be any one of these rows.

3.3 Mark test s and add it to $minCov$. Mark all entities covered by test s. Reduce $yetToCover$ by the number of entities covered by s.

`End of Procedure CMIMX`

Example 9.27 Suppose program P has been executed against the five tests in test suite T. A total of six entities are covered by the tests as shown in the following table; 0 (1) in a column indicates that the corresponding entity is not covered (covered). The entities could be basic blocks in the program, functions, definition-uses, or

any other testable element of interest. Further, a testable entity in P not covered by any of the five tests is not included in the table.

	1	2	3	4	5	6
t_1	1	1	1	0	0	0
t_2	1	0	0	1	0	0
t_3	0	1	0	0	1	0
t_4	0	0	1	0	0	1
t_5	0	0	0	0	1	0

Let us now follow procedure CMIMX to find the minimal cover set for the six entities. Input to the algorithm includes the 5×6 coverage matrix as shown in the above table.
Step 1: $minCov = \emptyset$. $yetToCover = 6$.

Step 2: All five tests and six entities are unmarked.

Step 3: As $yetToCover > 0$, we execute the loop body.

Step 3.1: Among the unmarked entities, 4 and 6 each contain a single 1 and hence qualify as the highest priority entities. Thus $LC = \{4, 6\}$.

Step 3.2: Among the unmarked tests, t_2 covers entities 1 and 4, and t_4 covers entities 3 and 6. Both tests have identical benefits of 2 each in terms of the number of entities they cover. We arbitrarily select test t_2. Thus $s = 2$.

Step 3.3: $minCov = \{2\}$. Test t_2 is marked. Entities 1 and 4 covered by test t_2 are also marked. $yetToCover = 6 - 2 = 4$.

Step 3.1: We continue with the second iteration of the loop as $yetToCover > 0$. Among the remaining unmarked entities, t_6 is the one with the least cost (=1). Hence, $LC = \{6\}$.

Step 3.2: Only t_4 covers entity 6 and hence $s = 4$.

Step 3.3: $minCov = \{2, 4\}$. Test t_4 and entities 3 and 6 are marked. $yetToCover = 4 - 2 = 2$.

Step 3.1: We continue with the third iteration of the loop as $yetToCover > 0$. The remaining entities, 2 and 5, have identical costs. Hence $LC = \{2, 5\}$.

Step 3.2: Entities 2 and 5 are covered by unmarked tests t_1, t_3, andt_5. Of these tests, t_3 has the maximum benefit of 2. Hence $s = 3$.

Step 3.3: $minCov = \{2, 3, 4\}$. Test t_3 and entities 2 and 5 are marked. $yetToCover = 2 - 2 = 0$.

Step 3: The loop and the procedure terminate with $minCov = \{2, 3, 4\}$ as the output.

A modified version of the greedy algorithm prioritizes the entities to be covered and then applies the greedy algorithm.

The difference between the greedy algorithm and `CMIMX` lies in Step 3.1 that is executed prior to applying the greedy test selection in Step 3.2. In this pre-greedy step, `CMIMX` prioritizes the entities for coverage. Entities covered by only one test get the highest priority, those covered by two tests get the next highest priority, and so on. One test that selects the highest priority entity is selected using the greedy algorithm. However, while the pre-greedy step allows `CMIMX` to find optimal solutions in cases where the greedy algorithm fails, it is not fail proof (see Exercises 9.31 and 9.32).

9.9 Test Prioritization

Test prioritization aims at ranking tests in a test suite using some cost criterion. It is then up to a tester to decide how many tests to use for regression starting from the test at the top of the list.

Given programs P and its modified version P', the various techniques for test selection discussed so far lead to a regression test suite T' derived from T used for testing P. While T' is a subset of T, it might be overly large for testing P'. One might not have sufficient budget to execute P' against all tests in T'. While test minimization algorithms could be used to further reduce the size of T', this reduction is risky. Tests removed from T' might be important for finding faults in P'. Hence one might not want to minimize T'. In such situations, one could apply techniques to prioritize tests in T' and then use only the top few high priority tests. In this section we introduce one technique for test prioritization.

Test with lower cost are given high priority as compared to tests with higher cost. Of course, one could use multiple criteria to prioritize tests.

Prioritization of tests requires a suitable cost criterion. Tests with lower costs are placed at the top of the list while those with higher cost at the bottom. The question obviously is: what cost criterion to use? Certainly, one could use multiple criteria to prioritize tests. It is important to keep in mind that tests being prioritized are the ones selected using some test selection technique. Thus, each test is expected to traverse at least some modified portion of P'. Of course, one could also prioritize all tests in T and then decide which ones to use when regression testing P'.

Residual coverage is one criterion used for prioritizing tests.

One cost criterion is derived directly from the notion of *residual coverage*. To understand residual coverage, suppose that E is a set of executable entities in P. For example, E could be any of set of basic blocks, definition-uses, all functions, or all methods in P. Suppose E is the set of all functions in P. Let $E' \subseteq E$ be the set of functions actually called at least once during the execution of P against tests in T'. We say that a function is *covered* if it is called at least once during some execution of P. Let us also assume that the library functions are excluded from E and hence from E'.

Let $C(X)$ be the number of functions that remain to be covered after having executed P against tests in set $X \subseteq T'$; initially $C(\{\ \}) = |E'|$. $C(X)$ is the residual coverage of P with respect to X.

The cost of executing P' against a test t in T' is the number of functions that *remain* uncovered after the execution of P against t. Thus $C(X)$ reduces, or remains unchanged, as tests are added to X. Hence the cost of a test is inversely proportional to the number of remaining functions it covers. Procedure `PrTest`, following the next example, computes the residual coverage for all tests in T' and determines the next highest priority test. This procedure is repeated until all tests in T' have been prioritized.

Example 9.28 Let $T' = \{t_1, t_2, t_3\}$ be the regression test for program P' derived from test set T for program P. Let E' be the set of functions covered when P is executed against tests in T' and $C(\{\ \}) = |E'| = 6$. Now suppose t_1 covers three of the six uncovered functions, t_2 covers two, and t_3 covers four. Note that there might be some overlap between the functions covered by the different tests.

The costs of t_1, t_2, and t_3 are, respectively, 3, 4, and 2. Thus t_3 is the lowest cost test and has the highest priority among the three tests. After having executed t_3, we get $C(\{t_3\}) = 2$. We now need to select from the two remaining tests. Suppose that t_1 covers none of the remaining functions and t_2 covers 2. Then the cost of t_1 is 2 and that of t_2 is 0. Hence t_2 has higher priority than t_1. After executing t_2, we get $C(\{t_2, t_3\}) = 0$. The desired prioritized sequence of tests is $< t_3, t_2, t_1 >$. Notice that execution of P against t_3 after having executed it against t_3 and t_1, will not reduce $C(X)$ any further.

`Procedure for prioritizing regression tests`

Input:
- T': Set of regression tests for the modified program P'.
- *entitiesCov*: Set of entities in P covered by tests in T'.
- *cov*: Coverage vector such that for each test $t \in T'$, $cov(t)$ is the set of entities covered by executing P against t.

Output: PrT: A sequence of tests such that (a) each test in PrT belongs to T', (b) each test in T' appears exactly once in PrT, and (c) tests in PrT are arranged in the ascending order of cost.

Procedure: `PrTest`

/*

PrT is initialized to test t with the least cost.The cost of each remaining test in T' is computed and the one with the least cost is appended to PrT and not considered any further. This procedure continues until all tests in T' have been considered.

*/

Step 1 $X' = T'$. Find $t \in X'$ such that $|cov(t)| \geq |cov(u)|$ for all $u \in X', u \neq t$.

Step 2 Set $PrT = < t >$, $X' = X' \setminus \{t\}$. Update *entitiesCov* by removing from it all entities covered by t. Thus $entitiesCov = entitiesCov \setminus$ cov(t).

Step 3 Repeat the following steps while $X' \neq \emptyset$ and $entityCov \neq \emptyset$.

3.1 Compute the residual coverage for each test $t \in T'$. $resCov(t) = |entitiesCov \setminus (cov(t) \cap entitiesCov)|$. $resCov(t)$ indicates the count of currently uncovered entities that will remain uncovered after having executed P against t.

3.2 Find test $t \in X'$ such that $resCov(t) \leq resCov(u)$, for all $u \in X', u \neq t$. If two or more such tests exist then randomly select one.

3.3 Update the prioritized sequence, set of tests remaining to be examined, and entities yet to be covered by tests in PrT. $PrT = append(PrT, t)$, $X' = X' \setminus \{t\}$, and $entitiesCov = entitiesCov \setminus cov(t)$.

Step 4 Append to PrT any remaining tests in X'. All remaining tests have the same residual coverage which equals $|entitiesCov|$. Hence these tests are tied. Random selection is one way to break the tie (also see Exercise 9.33).

```
End of Procedure PrTest
```

Example 9.29 Consider application P that consists of four classes C_1, C_2, C_3, and C_4. Each of these four classes is composed of one or more methods as follows: $C_1 = \{m_1, m_{12}, m_{16}\}$, $C_2 = \{m_2, m_3, m_4\}$, $C_3 = \{m_5, m_6, m_{10}, m_{11}\}$, and $C_4 = \{m_7, m_8, m_9, m_{13}, m_{14}, m_{15}\}$. In the following we refer to a method by an integer, e.g. m_4 by 4.

Suppose that regression test set $T' = \{t_1, t_2, t_3, t_4, t_5\}$. The methods covered by each test in T' are listed in the following table. Notice that method 9 has not been covered by any test in T' (see Exercise 9.26).

Test (t)	Methods covered ($cov(t)$)	$\|cov(t)\|$
t_1	1, 2, 3, 4, 5, 10, 11, 12, 13, 14, 16	11
t_2	1, 2, 4, 5, 12, 13, 15, 16	8
t_3	1, 2, 3, 4, 5, 12, 13, 14, 16	9
t_4	1, 2, 4, 5, 12, 13, 14, 16	8
t_5	1, 2, 4, 5, 6, 7, 8, 10, 11, 12, 13, 15, 16	13

Let us now follow procedure `PrTest` to obtain a prioritized list of tests using residual coverage as the cost criterion. The inputs to `PrTest` are: T', $entitiesCov = \{1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16\}$, and the test coverage vectors for each test in T' as in the table above. Note that we have excluded method 9 from

entitiesCov as it is not covered by any test in T' and hence does not cause a difference in the relative residual coverage of each test. Step 1: $X' = \{t_1, t_2, t_3, t_4, t_5\}$. t_5 covers the largest number of functions–13– in X' and hence has the least cost.

Step 2: $PrT = < t_5 >$. $X' = \{t_1, t_2, t_3, t_4\}$. $entitiesCov = \{3, 14\}$.

Step 3: We continue the process as X' and *entityCov* are not empty.

Step 3.1: Compute the residual coverage for each test in X'.

$$resCov(t_1) = |\{3, 14\} \setminus (\{1, 2, 3, 4, 5, 10, 11, 12, 13, 14, 16\} \cap \{3, 14\})| = |\emptyset| = 0$$

$$resCov(t_2) = |\{3, 14\} \setminus (\{1, 2, 4, 5, 12, 13, 15, 16\} \cap \{3, 14\})| = |\{3, 14\}| = 2$$

$$resCov(t_3) = |\{3, 14\} \setminus (\{1, 2, 3, 4, 5, 12, 13, 14, 16\} \cap \{3, 14\})| = |\emptyset| = 0$$

$$resCov(t_4) = |\{3, 14\} \setminus (\{1, 2, 4, 5, 12, 13, 14, 16\} \cap \{3, 14\})| = |\{3\}| = 1.$$

Step 3.2: t_1 and t_3 have the least cost. We arbitrarily select t_3. One may instead use some other criteria to break the tie (see Exercise 9.33).

Step 3.3: $PrT = < t_5, t_3 >$. $X' = \{t_1, t_2, t_4\}$, $entitiesCov = \emptyset$.

Step 3: There is no function remaining to be covered. Hence we terminate the loop.

Step 4: t_1, t_2, and t_4 remain to be prioritized. As *entityCov* is empty, the residual coverage criterion cannot be applied to differentiate among the priorities of these remaining tests. In this case we break the tie arbitrarily. This leads to $PrT = < t_5, t_3, t_1, t_2, t_4 >$.

Test prioritization does not eliminate tests; it merely ranks them. It is then up to a tester to determine how many tests from the list to use for regression testing.

Prioritization of regression tests offers a tester an opportunity to decide how many and which tests to run under time constraints. Certainly, when all tests cannot be run, one needs to find some criteria to decide when to stop testing. This decision could depend on a variety of factors such as time constraint, test criticality, and customer requirements.

Note that the prioritization algorithm `PrTest` does not account for any sequencing requirements among tests in T'. As explained in Section 9.2.3, two or more tests in T' may be required to be executed in a sequence for P, and also for P'. Exercise 9.35 asks you to derive ways to modify `PrTest` to account for any test sequencing requirements.

Test prioritization must account for sequencing constraints amongst tests.

9.10 Tools

Regression testing for almost any non-trivial application requires the availability of a tool. The mere execution of large number of regression tests can be daunting and humanly impossible under the given time and budget constraints. Execution of tests may require application and test setup, recording of the outcome of each test, performance analysis, test abortion, and several others.

There exist vela commercial and publicly available tools for regression testing. Few tools actually perform dynamic analysis or test prioritization.

A variety of commercial tools perform exactly the tasks mentioned above, and a few others. However, there exist few tools that offer facilities for test selection using static and dynamic analysis, test minimization, and test prioritization. While test execution is an essential and time consuming task in regression testing, test selection, minimization, and prioritization, when used carefully, could lead to a significant reduction in the cost of regression testing.

Next we briefly review three advanced tools for regression testing that feature one or more of the techniques discussed in this chapter. The tools we have selected have all been developed in commercial environments and tested on large programs. This is by no means a complete set of tools; there exist several others developed in research and commercial environments that are not reviewed here.

Table 9.1 summarizes the capability of three tools across several attributes. Both ATAC/χSuds and TestTube work on a given source program written in C. Echelon is different in that it does test prioritization based on binary differencing technique. A key advantage of using binary differencing, in contrast to source code differencing, is that it avoids the complex static analyses required to determine the nature and impact of code changes such as variable renaming, or macro definitions.

The strength of ATAC/χSuds lies in the assessment of test adequacy across a variety of criteria as well as a seamless integration of regression testing with test execution, slicing, differencing, code coverage computation, test minimization across a variety of coverage criteria, test prioritization, a combination of minimization and prioritization, and test management. The strength of TestTube lies in its use of a coarse coverage criterion for test selection, namely function coverage.

None of the tools described here have reported an application in the embedded domain. It is important to note that due to the hardware storage and application timing constraints, a regression testing tool that is satisfactory in a non-embedded application might be unsuitable in an embedded environment.

`Tool`: Selenium
`Link`: http://seleniumhq.org/
`Description`

Table 9.1 A summary of three tools to aid in regression testing.

Attributes	Atac/χSuds	TestTube	Echelon
Developer	Telcordia Technologies	AT&T Bell Laboratories	Microsoft
Year reported	1992	1994	2002
Selective retesting	Yes	Yes	No
Selection basis	Control/data flow coverage	Functions	Basic blocks
Test prioritization	Yes	No	Yes
Test minimization	Yes	No	No
Slicing	Yes	No	No
Differencing	Yes	Yes	Yes
Block coverage	Yes	No	Yes
Condition coverage	Yes	No	No
def-use coverage	Yes	No	No
p-use, c-use coverage	Yes	No	No
Language support	C	C	C and binary

Selenium is fast becoming a popular tool for regression and system testing of Web applications. The Selenium IDE is a Firefox add-on that allows simple record and playback of interactions with the browser. The Selenium WebDriver allows for the automation of regression tests. The WebDriver enables driving a Web browser using its native support for automation. Support is available for Java, C#, Python, Ruby, Perl, and PHP. Several browsers including Firefox, Chrome, and Opera are also supported.

`Tool`: SilkTest
`Link`: http://www.borland.com/us/products/silk/index.aspx
`Description`

This tool is available from Borland. This tool is a part of the Silk tool suite that provides tools for test management, test automation, integration with an IDE such as Eclipse, and performance testing. SilkTest enables automating tests for WEb-based and other applications.

SUMMARY

Regression testing is an essential step in almost all commercial software development environments. The need for regression testing arises as corrective and progressive changes occur frequently in a development environment. Regression testing improves confidence in that existing and unchanged code will continue to behave as expected after the changes have been incorporated.

While test execution and logging of the test results are important aspects of regression testing, we have focused in this chapter on techniques for test selection, minimization, and prioritization. These three areas of regression testing require sophisticated tools and techniques. We have described in detail the most advanced techniques for test selection using dynamic slicing. These techniques evolved from research in program debugging and have found their way in regression testing.

While test selection using some form of program differencing is a safe technique for reducing the size of regression tests, test minimization is not. Despite research in the effect of test minimization on the fault detection effectiveness of the minimized test sets, discarding tests based exclusively on coverage is certainly risky and not recommended while testing highly critical systems. This is where test prioritization becomes useful. It simply prioritizes tests using one or more criteria and leaves to the tester the decision on which tests to select. Of course, one might argue that minimization assists the tester in that it takes such decision away from the tester. ATAC/χSuds has a feature that allows a tester to combine minimization and prioritization and get the best of both worlds.

Exercises

9.1 In Example 9.2, we state "Under this assumption, when run as the first test, t_3 will likely fail because the expected output will not match the output generated by SATM'. " Under what conditions will this statement be false?

9.2 Given program P and test t, explain the difference between an execution trace of P against t when recorded as (a) a sequence of nodes traversed and (b) a set of nodes traversed. Use storage space as a parameter in your explanation. Give one example where a sequence is needed and a set representation will not suffice.

9.3 Draw the syntax trees in Example 9.7 not shown in Figure 9.6.

9.4 Write an algorithm that takes two syntax trees Tr and Tr' as inputs and returns true if they are identical and false otherwise. Recall from Section 9.5 that a leaf node might be a function name, say *foo*. In this case your algorithm must traverse the CFG of f, and the corresponding syntax trees, to determine equivalence.

9.5 In Example 9.9 there is only one function call at node 2 in `main` in Program P9.1. How will the application of `SelectTests` change if line 5 in `main` is replaced by: $\{p = g1(x, y) + g2(x, y)\}$?

9.6 Complete Example 9.9 to determine T' for the modified program.

9.7 Modify `SelectTests` such that T' includes only tests that traverse modified nodes in the main program and any called functions.

9.8 (a) Procedure `SelectTestsMainDecl` in Section 9.5.4 ignores the addition of any new variable declaration and deletion of any existing variable declaration. Does this imply that `SelectTestsMainDecl` is unsafe? (b) Suppose that variables added or removed from declarations in f are added to $declChange_f$. In what ways will this change in the definition of *declChange* affect the behavior of `SelectTestsMainDecl`?

9.9 List at least one advantage of regression test selection using the dynamic slicing approach over the plain modification traversing approach.

9.10 Suppose that the output statement at line 8 in program P9.5 is replaced by:

```
output (x, w);
```

Compute the dynamic slice with respect to x at line 8. Find the union of the dynamic slices computed with respect to variables x and w.

9.11 Consider test case t: $< x = 1, y = 0 >$, Program P9.5, and its modification. Will t be selected by the `SelectTests` algorithm?

9.12 Consider the following test t, and the values of $f1$, $f2$, and $f4$, for program P9.5.

t: $< x = 0, y = 3 >, f1(0) = 0, f2(0) = 4, f4(0) = 5$

Also assume that the output statement in Program P9.5 is as in Exercise 9.10. (a) Construct the DDG for Program P9.8 from its execution trace on t. (b) Compute the dynamic slices with respect to x and w at line 8. (c) Suppose P' is P9.8 obtained from P9.5 by adding two initialization statements for w and z and removing the `else` part from the `if` statement.

Program P9.8

```
1   input (x, y);
2   w=1;
3   z=1;
4   while (x< y){
5    if (f1(x)==0)
6      z=f2(x);
7    x=f4(x);
8    w=f5(z);
     }
9    output (x, w);
     end
```

Will t be included in T' when using the dynamic slicing approach for test selection? (d) What property must be satisfied by a test case for it not to be included in T' when using the dynamic slicing approach? Construct one such test case, construct its DDG, and compute the dynamic slice with respect to w at line 9 in P9.8.

9.13 Consider P to be Program P9.9.

Program P9.9

```
1   input (x,y);
2   z=0;
3   w=0;
4   while (x<y){
5    if (f(x)<0){
6     if (g(y)<0)
7      z=g(x*y);
8     w=z*w;
     } // End of first if
9    input (x, y);
    } // end of while.
10  output (w, z);
    end
```

Suppose now that P is executed against test t such that $trace(t) = (1, 2, 3, 4, 5, 6, 8, 9, 4, 10)$. (a) Construct the DDG for P from $trace(t)$. (b) Find the combined dynamic slice corresponding to variables w and z at line 10. (c) Compute the potential dependencies in the DDG. (d) Compute the combined relevant slice with respect to w and z.

9.14 Compute the relevant slice for program P9.6 corresponding to the trace obtained while executing it against t: $< N = 2, x = 2, 4 >$. Assume that $f(2) = 0$ and $f(4) > 0$.

9.15 Let P and P' be the original and the modified programs, respectively. Let T be the tests for P. Suppose that P' is obtained by modifying an output statement s in P. Suppose also that s is the last executable statement in P. Which tests in T will not be selected when using the execution slice approach? The dynamic slicing approach?

9.16 Refer to Examples 9.17 and 9.20 and let P be Program P9.6. Let e denote some error in P that is corrected to create P'. Assume that correction of e requires a change to one of the existing statements, or the addition/deletion of a statement. You may also change the specification so that P becomes erroneous. Now suppose that P is executed against test t and e is not detected. Construct an e that will go undetected if (a) $DS(t, z, 11)$ and (b) $RS(t, z, 11)$ are used for test selection.

9.17 Consider the modification of Program P9.5, as indicated in Example 9.21. Suppose that t: $< x = 1, y = 0 >$ and $t \in T$. Will t be selected for inclusion in T' by steps given in Section 9.6.6?

9.18 (a) Suppose that Program P9.9 is modified by removing a predicate as shown in Program P9.10.

Program P9.10

```
1   input (x, y);
2   z=0;
```

```
3   w=0;
4   while (x< y){
5    if (f(x)<0){      ← Predicate g(y) < 0 immediately following this line is removed.
6        z=g(x*y);
7      w=z*w;
     } // End of first if
8   input (x, y);
    } // end of while.
9  output (w, z);
    end
```

Should test t from Exercise 9.13 be included in the regression test for Program P9.10? (b) Now suppose that Program P9.10 is P and is modified to Program P9.9 by adding predicate $g(y) < 0$. Consider test t such that $trace(t) = \{1, 2, 3, 4, 5, 6, 7, 8, 4, 9\}$. Should t be included in the regression test for Program P9.9? (c) Give a general condition that can be used to decide whether or not a test should be included in T' when a predicate is added.

9.19 Let P be a program tested against test set T. Under one maintenance scenario, P' is derived from P through a series of modifications. These modifications are a mix of statement addition and deletion, predicate addition and deletion, addition of and removal of output variables, and perhaps others. Propose a systematic way of applying the relevant slice-based test selection technique that is scalable to large programs under the above maintenance scenario.

9.20 Dynamic slicing was originally proposed as a technique to aid in debugging programs. Subsequently it found an application in regression test selection. While the technique remains unchanged regardless of its application, the program dependence graph used as a source for computing the dynamic or relevant slice can be simplified when used for regression testing. Explain why.

9.21 Explain how one could use dynamic slicing as an aid in (i) program understanding and (ii) locating code corresponding to a specific feature implemented in a program.

9.22 Consider Program P9.7 in Figure 9.13. Suppose the following function trace $trace(t)$ is obtained by executing the program against test t: `main`, `f1`, `f2`, `main`. (a) Draw the dynamic dependence graph from $trace(t)$. (b) Now suppose a change is made in function `f3`. Will t be included in the regression test? (c) Will t be included in the regression test if a change is made to `f1`?

9.23 Compute the dynamic slice from the DDG in Figure 9.13 with respect to variable z at line 7.

9.24 Explain how you would compute potential dependencies in a DDG constructed out of function traces.

9.25 Exactly one node for each statement in the execution trace of P appears in an RDDG constructed using the procedure described in Section 9.6.8. While such an RDDG is useful for regression test selection, it looses some information needed for constructing the dynamic slice with respect to a variable when debugging P. What information is lost and how does the loss might affect debugging of P?

9.26 In Example 9.29, method 9 is not covered by any test in the regression test suite T'. Noting that $T' \subseteq T$, and T is the test set for application P, what might be the possible explanations for the lack of coverage of method 9 by tests in T'?

9.27 In what ways does a call graph differ from the dynamic dependence graph based on function trace?

9.28 Let T be a set of non-obsolete tests for P. Let P' be obtained by modifying P. Regression testing of P' is done using a subset T' of tests in T. What is the advantage of applying coverage-based test minimization procedure as in Section 9.8 to (a) T' and (b) T?

9.29 Let T be a test suite for testing P and T' a minimized test suite for testing the modified version P' of P. Give reasons why the fault detection effectiveness of T' might be less than that of T. Construct an example to show that indeed there exist cases where a fault in P' is detected by a test case in T but by none in T'.

9.30 What is the time complexity of the naive algorithm for finding the minimal test set? Suppose that program P is tested against n tests and a total of m entities are covered. Estimate how long it will take to find the minimal covering set when $n = 100, m = 300$; make suitable assumptions about the time it takes to perform various operations, e.g. set enumeration, required by the naive algorithm.

9.31 Consider the following "tricky" set cover problem. Let $E = \{1, 2, \ldots, p\}$ where $p = 2^{k+1} - 2$ for some $k > 0$. Let $TE = \{S_1, S_2, \ldots, S_k, X, Y\}$, be such that (a) S_i and S_j are disjoint for all $1 \leq (i,j) \leq k, i \neq j$, (b) S_i contains 2^i elements, (c) $S_1 \cup S_2 \cup \ldots \cup S_k = E$, and (d) X contains half the elements from each S_i and Y contains the remaining half. Thus $X \cap Y = \emptyset$ and $X \cup Y = E$. The problem is to find the minimal set cover of E.

For example, for $k = 2$ we have $E = \{1, 2, 3, 4, 5, 6\}$, $TE = \{S1, S2, X, Y\}$, $S1 = \{1, 2\}, S_2 = \{3, 4, 5, 6\}$, $X = \{1, 3, 4\}$, and $Y = \{2, 5, 6\}$. The minimal cover for this problem is $\{X, Y\}$. Show that both the greedy and the `CMIMX` algorithms fail to compute the minimal cover.

9.32 What is the importance of Step 3.1 in algorithm `CMIMX`?

9.33 In Example 9.29, two tests in Step 1 may tie for the minimum cost. A tie could also occur in Step 4 in procedure `PrTest`. Suggest at least two quantitative criteria that could be used to break the tie instead of breaking it through random selection as mentioned in procedure `PrTest`.

9.34 Let T' be a regression test for program P'. Let $X = \{< T'_1 >, < T'_2 >, \ldots, < T'_k >\}, k > 0$ be a set of sequences of tests such that each sequence $< T'_i >$ contains tests from T' and that each test in T' occurs in at least one sequence in X. Modify `PrTest` to account for the requirement that if any $< T'_i >$ is selected for execution, the given sequencing constraint is not violated. Note that the sequences in X might not be disjoint.

9.35 Given a program P composed of n functions and m tests, develop a formula to estimate the savings in storage requirements for function-level over statement-level execution traces. Make assumptions about the function size and coverage with respect to each test. How will your formula change for object-oriented programs where function-level execution trace is replaced by method-level execution trace?

10

Unit Testing

CONTENTS

10.1 Introduction
10.2 Context
10.3 Test design
10.4 Using JUnit
10.5 Stubs and mocks
10.6 Tools

The purpose of this chapter is to introduce tools and techniques used in unit testing. This phase of testing is often considered as the key to high quality software. There exist a variety of tools that aid the unit test process. Two such tools, namely, JUnit and EasyMock, are introduced. JUnit is a widely used tool for partially automating the unit test process. EasyMock allows mocking of objects that are not available while testing a unit. The technique of object mocking and its advantages are described using EasyMock.

10.1 Introduction

A program unit is a well defined piece of software that is combined with several others to create a software subsystem or a subsystem. As an example, a class or a collection of classes aimed to provide some utility or feature, is considered a unit.

Unit testing is an early phase of software testing. The goal in this phase is to ensure that each program unit functions as desired. A unit could be a class or a collection of classes in an Object-Oriented (OO) language such as Java or C#, or a function or a collection of functions in a procedural language such as C.

Units are generally coded by a single programmer in an effort to create a component specified in the design. A component itself might be a single unit or a collection of these. Together one or more components form a system. Thus, in unit testing the programmer who designed and coded the unit is likely to be responsible for creating test cases for the unit under test.

A unit is generally created by a programmer who also creates tests for it. In pair programming these tasks could be performed by two programmers.

Test cases for testing a unit could be created using a myriad of techniques described earlier in this book. For example, one could create tests using equivalence partitioning or from a finite state model of the unit's behavior. Further, the adequacy of unit tests can be assessed using any of the test adequacy techniques introduced in this book.

10.2 Context

Unit testing could occur at almost any stage in a software development process. It generally occurs whenever a unit is created.

Unit testing could occur in a variety of contexts. Figure 10.1 shows one possible context. It is assumed that the development process begins with an analysis of the requirements. The analysis provides information that is used in creating a design for the system under development. The design generally consists of one or more components that interact with each other to support system functionality. A high level design will generally correspond to the system architecture from which a more detailed design is derived.

Figure 10.1 One possible context in which unit testing may occur. The process will likely lead to multiple units and their tests though only one unit is shown here. Dotted links indicate continuation of the process for additional units.

Components in a large system will likely be developed by different development teams. The design of each component leads to the creation of distinct units of code such as a set of one or more classes or functions. The programmer may decide to begin coding the unit and then design the tests. However, when using the test-driven development methodology, one might first design the tests, as shown in Figure 10.1, and then code the corresponding unit. Note that given the requirements, it is possible to design the tests prior to coding a unit.

Large software systems are often designed by several people and consist of multiple units. Each programmer is responsible for the unit designed by her/him.

Once the unit has been coded, it is tested against the tests derived. In the event the unit under test is complex, one might use incremental development. In that case, testing will proceed in a loop where the unit developed so far is tested until deemed correct and then enhanced and tested again. Thus, the three steps, namely, test design, coding, and testing will likely be performed several times in a loop as shown.

Test adequacy assessment is done after the unit has passed all tests. Adequacy assessment may be an internally imposed requirement or imposed by the client. Code coverage measures such as decision coverage or MC/DC coverage are often used as satisfactory adequacy criteria. If tests are found inadequate with respect to the adequacy criteria considered, new tests need to be designed and the unit retested. The unit test process is considered completed when the unit has passed all tests that meet the required adequacy criteria.

Adequacy of unit tests is best done when the unit has passed all tests. Lack of adequacy may lead to the addition of new tests.

It is important to note that the above description is one of the several ways in which unit testing could be embedded in the development process. For example, as an alternative to the above mentioned process, one might design tests after having coded a portion of the unit. The development and test process could then continue in an incremental manner until some test completion criteria are satisfied.

10.3 Test Design

A test case is a set or a sequence of inputs to a unit under test together with the expected response. As mentioned earlier, one could derive test cases using one or more test generation or enhancement methods. When test cases are generated prior to coding the unit, one needs to use a black-box method, e.g. equivalence partitioning. The W-method could be used when the expected behavior of the unit can be modeled as a finite state machine.

A test case can often be coded as a program. Upon execution this program supplies inputs to the program under test. Coded test cases also allow easy re-execution of the program against a test.

In some situations, the input data in a test case derived from the requirements could be input to the unit manually. However, it is often more efficient to write a program that encapsulates the test case, both the inputs

A unit testing tool, such as JUnit, allows coding of tests and automated execution of a program against these tests.

and outputs, invokes the unit with the input data, and checks if the response is as expected. Such coding could be simple or complex depending on the complexity of the unit under test. In any case, coded test cases allow for ease of re-execution when there is a change in the unit. Tools such as JUnit allow for coding multiple test cases and automatically execute the unit against these. The next example demonstrates generation of tests for a unit using equivalence partitioning of the output domain. Coding of these tests in JUnit is the topic of the next section.

Example 10.1 One well-known testing problem considers testing a program that classifies a triangle. The problem is to generate an adequate set of test cases passing which will ensure that the classification program is correct.

The classifier is assumed to be coded to meet the following requirements. The triangle classifier program takes three inputs, a, b, and c, representing lengths of the three sides of a triangle. It must first check whether or not the sides make a triangle. If they do not then an exception is raised, else the correct triangle type is returned. Let us assume that the following types of triangles are considered: right angle (1), equilateral (2), isosceles (3), and scalene (4).

Test cases can be derived by partitioning the output domain.

Let us derive test cases based on partitioning the output domain, i.e. the range, of the classifier based on the above requirements. Thus, five test cases are needed. The exact values of the lengths may differ but one sample follows.

Test 1: To test if an exception is raised when the input is invalid.
Input: $< a = -1, b = 1, c = 1 >$
Expected response: Exception "Invalid input" must be raised.

Test 2: To test for a right-angled triangle.
Input: $< a = 4, b = 4, c = 5.65 >$
Expected response: 1

Test 3: To test for an equilateral triangle.
Input: $< a = 1, b = 1, c = 1 >$
Expected response: 2

Test 4: To test for an isosceles triangle.
Input: $< a = 1, b = 1, c = 2 >$
Expected response: 3

Test 5: To test for a scalene triangle.
Input: $< a = 1, b = 2, c = 3 >$
Expected response: 4

In test-driven development, the above tests would be coded using a tool, for example JUnit, and executed against a classifier that simply returns, say 0, regardless of the input. Thus, the classifier will fail all test cases. One may now incrementally develop the classifier and test each increment. Gradually more and more tests will pass until all tests are executed (see Exercise 10.4).

10.4 Using JUnit

Execution of unit tests can be automated through tools such as JUnit for Java and NUnit for all .Net languages such as C#.

Several tools are available for automating the unit test process. JUnit is a popular tool for unit testing of Java programs. Once the tests have been created as, for example, described in the previous section, they can be coded in Java as JUnit tests. JUnit can automatically execute the unit under test against all tests. For each test it informs the tester whether the test passed or failed. There are several ways to create a JUnit test or a collection of JUnit tests. Below we describe one possible way to test a class using JUnit. Note that this section is intended to expose you to JUnit and is not a full length tutorial. Details of JUnit can be found by following the link mentioned in Section 10.6.

JUnit allows grouping of multiple test into a class.

A JUnit test case is a class that extends `TestCase`. Within a JUnit test case there are several tests each named `testX`, where `X` denotes the specific feature to be tested. For example, `testEquilateral` denotes a test for equilateral triangle. JUnit does not care what `X` is, but it is useful to give mnemonics to each test.

When executing the test case, JUnit first calls method `setUp()`. This method is optional and if included in the test case it sets up the test. For example, the `setUp()` method might create an object from the class under test and pass some initialization parameters. JUnit can be made to call `setUp()` after each test, or once for all tests are executed in a test case.

Next, JUnit runs all tests, as mentioned earlier each test is coded as a method named `testX`. Finally, the `tearDown()` method is called. This method is intended to perform activities needed to clean up any changes that the tests might have made in, for example, a database. Essentially, when needed, `tearDown()` restores the state of the system that exists prior to the execution of a test or after the execution of all tests. The next example illustrates the use of JUnit to test the triangle classifier program using test cases derived in the previous section.

Example 10.2 Let us code the test cases derived in the previous section as a JUnit test case. Clearly, we need five tests. The following class contains these five tests named `testInvalid`,

`testRightAngled`, `testEquilateral`, `testIsosceles`, and `testScalene`. In addition there is a `setUp( )` method that creates a `Triangle` object and passes to it the acceptable error bound when checking for a right-angled triangle.

Listing 10.1 A JUnit test case for the triangle classification program. (TriangleTest . java)

```
import org . junit . Test;
import org . junit . BeforeClass ;
import static org . junit . Assert . ∗ ;
import org . junit . Ignore ;
import junit . framework . ∗ ;
/ / Expected Results :
/ / Exception – invalid input . 1 – right angle . 2: equilateral .
/ / 3 – isosceles . 4 : scalene
public class TriangleTest extends TestCase {
   double maxError ;  / / Max tolerance for right triangle test .
  double a , b , c ;  / / Lengths of sides .
  int result ; / / response form the classifier
  Triangle t ;  / / Triangle object.
  @BeforeClass
  public void setUp ( ) {
     t = new Triangle ( 1 . 0E–1 ) ;
  } / / End of setUp ( )
  @Test ( expected=InvalidInputException.class )
  public void testInvalid ( ) {
    a = – 1 ; b = 1 ; c = 1 ; / / Set length of sides
    result=t . classify (a , b , c ) ; / / Invoke classifier
    assertEquals ( 0 , result ) ; / / Check if result is as expected
  } / / End of testInvalid ( )
  public void testRightAngled ( ) {
    a = 4; b = 4; c = 5 . 65;
    result = t . classify ( a , b , c ) ;
    assertEquals ( 1 , result ) ;
  } / / End of testRightAngled ( )
  public void testEquilateral ( ){
     a = 1 ; b = 1 ; c = 1 ;
     result = t . classify ( a , b , c ) ;
     assertEquals ( 2 , result ) ;
  } / / End of testEquilateral ( )
  public void testIsosceles ( ) {
```

```
      a = 1 ;  b = 1 ; c = 2 ;
      result = t . classify (a , b , c) ;
      assertEquals (3 , result) ;
    } / /  End of testIsosceles ( )
    public void testScalene ( ) {
       a = 1 ;  b = 2 ; c = 3 ;
       result = t . classify (a , b , c) ;
      assertEquals (4 , result) ;
    } / / End of testScalene ( )
  } / /  End of TriangleTest
```

The `TriangleTest` class extends `TestCase` indicating that this class contains a collection of JUnit tests. Each test defines the three lengths created as test cases in the previous section. It then calls the `classify()` method with the lengths as the parameters and captures the response in `result`. The `assertEquals()` method compares the value of `result` with the expected value. For example, `assertEquals(1, result)` checks if `result` is equal to 1.

When run, JUnit executes each test and displays the outcome using color coded names of the tests. For example, following is the output (not in color) from JUnit after applying the five tests above to a sample triangle classification program.

While unit tests can be coded using JUnit, they need to be designed using some other process. For example, tests could be constructed manually and then coded and grouped using JUnit for automated execution.

Stubbing and mocking are two techniques for use in testing OO programs.

```
1 warning found:
  TriangleTest
    testInvalid
    testRightAngled
    testEquilateral
    testIsosceles
    testScalene
File:/Users/apm/Desktop/SecondEdition/Chapters/Part-II-Test
Generation/req-chapter/Programs/mockObjects/TriangleExample/
TriangleTest.java [line: 21]
Error: InvalidInputException: Invalid input.
```

All tests except `testInvalid` are displayed in green indicating that they passed. The warning generated by `testInvalid` is displayed after a listing of all the tests. (Also see Exercises 10.4 and L10.6.)

Figure 10.2 Class CC is to be tested. CC uses class CU. (a) Stubbing CU when CC is not available. (b) Mocking CU using the inversion control pattern when CU is not available.

JUnit offers several annotations to manage the tests. `@Before` placed before `setUp` indicates that the set up is to be executed after each test. `@BeforeClass` indicates that `setUp()` is to be executed only once and prior to the execution of any test. `@Test` indicates that the following method is a test. Several other annotations are available and can be found in the JUnit reference guide.

Management of test execution is done in JUnit using annotations such as `@Before` and `@Test`.

10.5 Stubs and Mocks

A mock object can be used in place of a stub. Consider the case where unit A needs to be tested. Unit A needs unit B as a server. Also assume that unit B is not yet available. In such a situation, one can replace B by a stub or by a mock object. Figure 10.2 is a visual description of how mock objects are used.

There are some key differences between what is expected of a stub and of a mock object. A stub serves as a simplistic replacement of an unavailable unit. Thus, a stub generally provides simplistic implementation of the unavailable object. When a request is sent to a stub, it might return a predefined value instead of computing it or connecting to a database. The values returned from a stub are then checked for correctness and used in further testing. This is also known as *state verification*.

Another form of verification is known as *behavior verification*. In this case the unit is tested to check that its interaction with the server is as expected. Thus, for example, in addition to testing whether or not B returns the correct value, it is also checked whether or not A calls the methods in the server in the correct order. Mock objects are useful where behavior verification is needed.

A stub or a mock replace an object that is not available during a test. A stub is generally used for state verification while a mock for behavior verification.

When using tools such as JUnit, the test for a unit uses an `assert` statement to check whether or not a value returned by the client is correct. This is an example of state verification. However, an `assert` statement does not verify whether or not the unit under test has called the correct methods in the client and in the correct order. This form of checking is an example of behavior verification and can be done using mock objects.

For state verification JUnit offers the "assert" statement to check whether or not a value returned by by a unit under test is correct.

10.5.1 Using mock objects

To understand how mock objects and `EasyMock` can be used, suppose that class `A` (client) is under test. The class testing `A` is named `TestA`. Suppose also that `A` uses class `B` (server) but that `B` is not available at the time of testing `A`. Let `IB` be an interface to which `B` is to be implemented. Let `compute()` be an abstract method in `IB` that takes an integer as input and returns an integer.

Figure 10.3 Sequence of actions when using the `EasyMock` tool for unit testing. A sample sequence of actions for one test is shown. `A` is a client class under test that uses server class `B` that implements interface `IB`. The mock object mocks `B` and allows `A` to be tested without `B`.

EasyMock is an open source tool used to automate testing using mock objects. It can be used to create mock objects that simulate the behavior of the object mocked.

There are several tools to help automate the process of creation and use of mock objects. `EasyMock` is one such tool. The following steps, also shown in Figure 10.3, lead to a test using a mock object instead of using a stub or directly using `B`. Here it is assumed that the client needs only one server that is not available at the time of testing. The steps below can be extended to create multiple mock objects if needed.

1. `TestA` requests `EasyMock` to create a mock object. It supplies the name of the interface class (`IB` in this example) to `EasyMock`.
2. `EasyMock` creates a mock object capable of simulating the behavior of an object created from a class that conforms to the interface (`IB`).
3. `TestA` requests `EasyMock` to set the behavior expected of the mock object. For example, when method `compute()` is invoked on `B` with input 4, then the return value must be 5.

4. `EasyMock` makes a note of all expectations of `B`. Note that while the expectations are of `B`, the mock object is intended to simulate `B` and hence needs to know the expectations.
5. `TestA` sets the mode of `EasyMock` to *replay*. In this mode `EasyMock` is waiting to receive calls from the client and respond to them as expected.
6. `EasyMock` enters the *replay* mode.
7. `TestA` passes a reference to the mock object to the client under test (`A`).
8. The client saves the reference to the mock object.
9. `TestA` makes a call to the client to test a feature.
10. While processing the call from `TestA`, the client invoked a method in `B`. However, this call is processed by the mock object using the expectations provided earlier.
11. The client returns from the call. The return data depends on the method under test in the client. It also likely depends on the value returned by the mock object in response to the call from the client in the previous step. The previous and this step can be repeated as many times as needed to perform the test.
12. Finally, `TestA` makes a call to `EasyMock` to verify if the client behavior is as expected.

In its "replay" mode, EasyMock waits to receive calls from clients and responds to these as expected.

There are several variations of the sequence explained above. The following example illustrates the above steps in the context of `JUnit` and a tool named `EasyMock`.

Testing using mock objects is facilitated by tools such as JUnit and EasyMock.

Example 10.3 Suppose that class `Compute` performs computations on an integer. However, due to the complexity of the computation and the possibility that the algorithm may change, `compute` makes use of class `Expert`. Methods `calc()` and `save()` are provided by `Expert`; `calc()` performs the desired computation on `x` and returns the results while `save()` saves `x` and the result in a database.

`Compute` takes two inputs: `request` and `x`. If `request=1`, it invokes `calc()` and returns the result to the user. If `request=2`, it invokes `calc()` followed by `save()`. An exception is raised if the request is invalid.

It is desired to test `Compute` when `Expert` is not available. The test can be performed using a stub in place of `Expert`. However, for reasons described earlier, a mock object can be used in place of `Expert`. Let us follow the steps described earlier to create a few tests using the following interface to which the `Expert` as well as

the mock object are created. `ExpertInterface` given below is the class to which `Expert` is written.

Expert interface The code for the interface to which `Expert` is written follows. The signatures of the two methods provided by `Expert` are at lines 6 and 7.

Listing 10.2 Expert interface. (ExpertInterface . java)

```
1  /*
2   * Interface of the class that is not yet available
3   * when class Compute is to be tested.
4   */
5  public interface ExpertInterface{
6      public int calc (int x) ; // Calculate
7      public void save (String s) ; // Save
8  }
```

Class under test
The class under test, namely `Compute`, uses an object designed to `ExpertInterface` to actually perform the calculations and save the results. The code follows.

Listing 10.3 Class to compute and save. (Compute . java)

```
1   /*
2    * This class provides computation service.
3    * It uses another object written to
4    * ExpertInterface to actually comppute
5    * and save the result in a database.
6    */
7   public class Compute{ // Under test
8       ExpertInterface expertObject;
9       public void setExpert ( ExpertInterface expertObject ) {
10          this . expertObject = expertObject ;
11      } // End of setExpert
12      public int compute ( int request, int x ) {
13          if (request = = 1 )
14              return (expertObject . calc ( x ) );
15          if ( request = = 2 ) {
16              int result = expertObject . calc ( x ) ;
17              save ( x, result ) ;
18              return ( result ) ;
```

Listing 10.3

```
19          }else {
20              throw new InvalidCodeException ( " Invalid request " ) ;
21          }
22      }// End of compute ( )
23      public void save ( int x , int result ) {
24          expertObject . save ( " For " + x + " computation result :
                " + result ) ;
25      }// End of save ( )
26  }// End of Compute
```

At line 10, the `setExpert` method saves the reference to the expert object for future use. Note that when the actual expert object is not available, this would be a reference to the mock object. Method `compute` takes in `request` and `x` as inputs and invokes the appropriate sequence of methods to perform the computation and, if `request=2`, then saves the result. Note that the `calc` method is invoked at lines 14 and 16 on the expert object, a reference to which was saved earlier in `setExpert`. Thus, `compute` does not really know whether the requested computation is being performed by the real expert or by its mock.

`JUnit` *tests*

One could create a variety of JUnit tests to test the methods in `Compute`. Below are four tests to show a few capabilities of `EasyMock`.

Many tests can be created in JUnit to test methods inside an object. The adequacy of such tests can be assessed using a variety of coverage criteria.

Listing 10.4 A set of four JUnit tests for Compute. (TestCompute . java)

```
1   /*
2    * Test Compute using a mock object.
3    */
4   import junit . framework . TestCase ;
5   import org . junit . Before ;
6   import org . junit . Test ;
7   import org . easymock . EasyMock ;
8   /* *
9    * A JUnit test case clas.
10   * Every method starting with the word "test" is called
11   * when running the test with JUnit .
12   */
```

Listing 10.4

```
public class TestCompute extends TestCase {
    private ExpertInterface mockExpert ;
    private Compute calcSave ;
    @Before
    public void setUp ( )    {
        mockExpert=EasyMock.createStrictMock (
          ExpertInterface . class ) ;
        calcSave = new Compute ( ) ;
        calcSave . setExpert(mockExpert) ;
    }// End of setUp ( )
    // Check if method returns correct value .
    public void test 1 ( )    {
        EasyMock . expect ( mockExpert . calc ( 5 ) ) . andReturn ( 50 ) ;
        EasyMock . replay ( mockExpert ) ;
        assertEquals ( 50 , calcSave . compute ( 1,5 ) ) ;
    }// End of test1 ( )
    // Check number of times called .
    public void test 2 ( )    {
        EasyMock . expect ( mockExpert . calc ( −5 ) ) . andReturn ( 0 ) .
          times ( 2 ) ;
        EasyMock . replay ( mockExpert ) ;
        asser tEquals ( 0 , calcSave . compute ( 1 , −5));
        asser tEquals ( 0 , calcSave . compute ( 1 , −5));
    }// End of test 2 ( )
    // Check method sequencing .
    public void test 3 ( )    {
        EasyMock . expect(mockExpert . calc ( 5 ) ) . andReturn ( 4 ) ;
        mockExpert . save ( ( String ) EasyMock . anyObject ( ) ) ;
        EasyMock . replay ( mockExpert ) ;
        assertEquals ( 4 , calcSave . compute ( 2 , 5 ) ) ;
        EasyMock . verify ( mockExpert ) ;
    } // End of test3( )
      // Check invalid parameter .
    @Test ( expected = InvalidCodeException . class)
    public void test 4 ( )    {
        EasyMock . expect ( mockExpert . calc ( 5 ) ) . andReturn ( 4 ) ;
        EasyMock . replay ( mockExpert ) ;
        assertEquals ( 4 , calcSave . compute ( 3 , 5 ) ) ;
    }
}
```

Mock objects could be "nice" or "strict."

The JUnit test contains five methods. Method `setUp` is invoked prior to the start of a test. It creates a mock object using `easyMock` at line 18. Note that a *strict* mock object is created using the `createStrictMock( )` method. One could also create a non-strict as well as a *nice* mock object. The type of mock object to create depends on what aspects of the class under test are to be tested. Strict mock is used in this example as we wish to test the order of method calls.

After creating a mock object, the `setUp` method creates a `Compute` object at line 19 and passes it at line 20 to the mock object just created. This is necessary so that during testing, calls to `Expert` are routed to the mock object.

The four tests are named `test1` through `test4`. The overall structure of each test is nearly the same. First, the expectations from the mock object are defined. This is done using the static `expect` method in `EasyMock`. For example, at line 24 in `test1`, `EasyMock` is informed that when called with 5 as a parameter, `calc` should return 50. In `test2` at line 30, we inform `EasyMock` that when `calc` is invoked with -5 as a parameter it should return 0. This line also specifies that `calc` must be invoked twice before any other method is invoked. Note that the expectations of a mock object are derived from the specifications of the class that is being mocked.

After having set the expectations, `EasyMock` is switched to *replay* mode using the `replay` method at line 25. In the replay mode `easyMock` simply "watches" the test process and makes a note of the events. At line 26 the `compute` method is invoked with `request=1` and `x=5`. The expected return value is 50 as indicated at line 24. The JUnit `assert` command is used to check whether or not the return value is correct.

In `test2` it is assumed that `compute` must be invoked twice with -5 as the value of `x`. The test will fail if this does not happen. At line 26 `test3` checks if `compute` and `save` are invoked in a sequence. The sequence expected is set at lines 37 and 38. Recall that when called with `request=2`, `compute` must invoke `calc` and `save` on the `Expert` object. At the end of the calls the `verify` method in `EasyMock` checks whether or not the expected sequence did occur during the test.

`EasyMock` can be used to check if an exception is raised when expected. This is illustrated in `test4`. The expectation in terms of parameters and the return value is set at line 46. The expectation that an exception is to be raised is set using annotation

`@Test` provided by JUnit. The test occurs at line 48. As written in this example, all tests pass except `test4`. The code for `Invalid CodeException` follows. Also, see Exercises 10.9 through 10.11.

Listing 10.5 Exception raised when request is not recognized. (InvalidCodeException.java)

```
1  public class InvalidCodeException extends
       RuntimeException {
2      public InvalidCodeException ( String message ) {
3          super ( message ) ;
4      }
5  }
```

10.6 Tools

A number of tools are available for unit testing, and for a variety of programming languages. Many of these tools also offer some form of code coverage measurement.

Several tools are available for generating tests using techniques introduced in this chapter or similar ones. Some of these tools are listed below with links to where they could be found.

`Tool`: JUnit
`Link`: http://pyunit.sourceforge.net/
`Tool`: PyUnit
`Link`: http://pyunit.sourceforge.net/
`Tool`: SpartanUnit
`Link`: http://openhopla.codeplex.com
`Description`

The above three tools are intended for use during unit testing. JUnit is for testing Java programs, PyUnit for Python programs, and SpartanUnit for C# programs. These are not test generation tools. Instead, they allow a tester to code their tests. The program under test is then executed automatically against all the coded tests. Thus, these tools aid in reducing the time to set up tests and execute them each time a change is made in the program under test. At the time of this writing, SpartanUnit, developed by John Spurgeon, is a framework for constructing new testing tools rather than a full fledged tool such as JUnit.

SUMMARY

This chapter describes some aspects of unit testing. The context of unit testing is explained. An example is used to illustrate how the myriad of test generation techniques can be used to generate tests during unit testing. How to code and automate the execution of units tests is explained using JUnit which is a popular test automation tool for programs written in Java. The test-driven development technique is also explained with an example.

Testing a unit often requires the availability of other collaborating units. However, when a collaborator is not available, one could either replace it with a stub or instead use a mock object. While both approaches are used in practice, OO developers sometimes use tools such as EasyMock to automatically generate mock objects. The use of mock objects in unit testing is explained with a detailed example.

Exercises

10.1 What would you consider as a "unit" in a software system? Is it necessary for a unit to be a relatively small piece of code, or a relatively large component could also be considered as a unit?

10.2 Why is it important to perform unit testing? How is unit testing different from integration testing? [You may return to this question after having read Chapter 11.]

10.3 There are two loops in Figure 10.1. Explain why the steps in these loops might be executed more than once.

10.4 Example 10.2 does not list the triangle classifier program for which the `TriangleTest` class is designed. Write a class named `Triangle` that meets the requirements in Example 10.1. Then run the JUnit tests in `TriangleTest`. Develop the code incrementally. Run `TriangleTest` against each increment. Continue the process until all tests have passed.

10.5 Figure 10.1 shows unit testing as an activity following system design. In what other contexts might one need to test one or more units in a system?

10.6 Use a code coverage tool to assess the adequacy of tests derived in the previous exercise. EMMA is one easy-to-use coverage analyzer for Java. It computes class, method, block, and statement coverage. Enhance your tests if not found adequate with respect to each of the four coverage values reported by EMMA.

10.7 Give an example of a scenario where the `setUp()` method should be called only once before all the tests are executed and not each time before a test is executed.

10.8 Give one example of a scenario where the `tearDown()` method be called after all tests have been executed.

10.9 (a) List a few advantages of using a mock object in place of a stub. (b) Is it necessary for the mock object to be created using an interface? (c) Mention the differences between `createMock()`, `createStrictMock()`, and `createNiceMock()`.

10.10 To be able to complete the exercises related to `EasyMock`, you will need to download its library and make sure that it is in JUnit's path.

Let us now play with the code given in Example 10.3. (a) Select any passing test and try to make it fail. (b) Modify the `ExpertInterface` to add the following method.

```
public double enhancedCompute(double x)
```

Now modify `compute()` in class `Compute` so that when `request=3`, `enhancedCompute` must be called followed by `save`. (c) Assume that when `request=4`, `enhancedCompute()` is called twice. In the first call the parameter `x` is input. In the following call the value returned from the first call is input. These two calls are followed by a call to save the final result. Now create a test to check if this call sequence was exercised. (d) Suppose that when `x<0`, `enhancedCompute()` raises the `InputOutOfRangeException` exception. Create a test to check this requirement.

10.11 Class `CurrConvert` contains three methods: `getCountries()`, `getAmount()`, and `convert()`. Method `getCountries()` asks the user to select countries C_1 and C_2, where the currency of C_1 is to be converted to that of C_2. Method `getAmount()` gets the amount in the currency of C_1. Method `convert()` converts the given amount into the equivalent currency of C_2. For conversion it uses another class named `CurrData` that contains a method `getRate()` to obtain the latest conversion rate between the currencies of C_1 and C_2.
Code `CurrConvert` in a language of your choice. Assume that `CurrData` is not available when testing `CurrConvert`. Write an interface `CurrConvertInterface`. Using JUnit, or any other unit testing tool available for the language of your choice, create and run tests for `CurrConvert` using a mock object for `CurrData`. Try using equivalence partitioning and boundary value analysis techniques to generate the tests.

11

Integration Testing

CONTENTS

1.1 Introduction
1.2 Integration errors
1.3 Dependence
1.4 OO versus non-OO programs
1.5 Integration hierarchy
1.6 Finding a near-optimal test order
1.7 Test generation
1.8 Test assessment
1.9 Tools

The purpose of this chapter is to introduce the objectives of integration testing and the techniques used. Integration testing occurs when two or more modules, components, or subsystems are combined to form a new subsystem or a system. During the integration testing phase it is generally assumed that the components being integrated have been tested independently. Hence, the focus in this phase is on errors that may exist due to communication among components being integrated. Often, when many components are being integrated, one needs to determine their integration sequence. Three algorithms to determine an optimal or a near-optimal integration sequence are described and compared in this chapter.

11.1 Introduction

Integration testing occurs when two or more units are combined and tested. In this phase of testing it is assumed that the units combined have already been tested. Hence, the primary intention during integration testing is to find whether or not a subsystem consisting of multiple units works as desired.

A unit (also referred to as a component) could be as simple as a class in Java, or as complex as a database server.

This chapter is concerned with issues that relate to integration testing. This type of testing refers to a testing activity wherein two or more components of a software system are tested together. A software component is sometimes also referred to as a class, a collection of classes, a module, a collection of methods, and so on. In integration testing a component might be as simple as a class in Java, or a file consisting of methods in C. Or, it could be as complex as a collection of many Java or C++ classes or several files each containing multiple methods in C.

Goal of integration testing: The goal of integration testing is to discover any problems that might arise when the components are integrated. Thus, for example, components C1 and C2 might have been tested and found to function correctly. However, this does not guarantee that when integrated, C1 and C2 will perform as expected. An error discovered during integration testing is often referred to as *integration error*. Hence the goal of integration testing is to discover any integration errors when two or more components are combined into a system or a subsystem. Of course, during integration testing one might discover an error in one of the components that had been tested previously. However, integration errors are those that would likely be revealed only when the components are integrated and tested together.

When does integration testing occur? Integration testing occurs at various levels. A developer might code a class C1 and test it. The same developer might then code class C2 that uses C1, and test both C1 and C2 as an integrated unit. This process may continue as the developer moves toward the goal of creating a reasonably complex component that consists of two or more classes. At a much higher level, a team of testers might integrate two or more complex components, or subsystems, perhaps developed by different teams within an organization. In yet another case, such a team of testers might integrate and test several components developed in-house with one or more components obtained from different sources.

When more than two components are to be integrated one needs to determine the order of integration and testing.

Regardless of the level at which integration testing occurs, one faces a set of problems that ought to be solved prior to initiating the test. The following questions that arise during integration testing are considered in the remainder of this chapter.

A stub is a place holder for a component that is not available during integration testing.

1. *Integration hierarchy*: Should the components be integrated top-down, bottom-up, or using a mix of these two approaches?
2. *Sequencing and stubbing*: In what sequence should the components be integrated and what stubs are needed?

3. *Test generation*: What methods should be used to generate tests and to measure their adequacy?

In the remainder of this chapter, the terms *class*, *component*, and *module* are used synonymously unless differentiated explicitly.

Integration tests focus on the connections across components. Hence, the aim is to detect errors that may lead to incorrect communication among the components.

11.2 Integration Errors

During integration testing, it is assumed that the components being integrated have been tested well, though independently. Thus, integration tests focus on the connections across the components being integrated. It is therefore useful to know what types of errors one might encounter during integration testing. Despite the fact that components being integrated have been tested, they might contain errors the impact of which is visible only when the components are integrated. Such errors are known as integration errors. Of course, one cannot deny the possibility that such errors may be found in unit testing. However, integration testing offers yet another opportunity to discover such errors.

Integration errors could be classified into several categories; four such categories are considered in this chapter.

Figure 11.1 shows the consequence of four types of integration errors. The figure offers a high level classification of the consequence of errors in components being integrated. Indeed, some of the errors that lead to

Figure 11.1 Types of integration errors. D_I is a set or a sequence of inputs passed from component `C1` to component `C2`. D_O is a set or a sequence of outputs returned from component `C2` to component `C1`.

the incorrect values shown in Figure 11.1 should have been found in unit testing. However, when the units are tested with stubs, incorrect values at the interfaces might go unchecked and revealed only during integration testing.

Type A behavior in Figure 11.1 is a consequence of an error in component C1 and perhaps also in component C2. Type B behavior extends Type A by assuming that component C1 responds incorrectly and consequently results in an incorrect response from component C1. Type C behavior implies an error in component C2 and perhaps also in C1. Lastly, Type D behavior implies an error in C2.

Example 11.1 Consider the errors that may lead to the four types of behaviors exhibited in Figure 11.1. Events in the figure are numbered indicating the sequence in which they occur. Suppose that a call from C1 to a method in C2 sends incorrect parameter values (event 1). In turn, C2 computes incorrect results (event 2). This error in C1 leads to Type A behavior. Type B behavior results from a similar error but when C1 receives an incorrect result from C2 (event 2) it also generates an incorrect output (event 3). In either of these two cases, it is clear that C1 has an error but C2 might or might not.

Type C behavior could occur due to any one of several types of coding errors that lead to an incorrect output that in turn leads to an incorrect output from C1. Type D behavior is similar to Type C except that the incorrect output of C2 does not affect C1.

The categorization of different interface behaviors as in Figure 11.1 during integration testing is useful when assessing the adequacy of integration tests. Such adequacy assessment, using a powerful technique known as *interface mutation*, is discussed briefly in Chapter 8.

Components combined to form a system or a subsystem exhibit various kinds of dependencies. Three such forms are: data dependence, functional dependence, and control dependence.

11.3 Dependence

Classes and collections of classes, better known as components, to be integrated are often related to each other leading to inter-class or inter-component dependencies. For example, class C1 might generate data and write it to a file. Class C2 might read the data written by C1 and perform further processing. In this example, C2 depends on data generated by C1 and hence we say that there exists *data dependency* between C1 and C2. In another case, C1 might make use of a function in C2 thus resulting in *functional dependency*. In yet another case, C1 might evaluate

a condition that affects whether or not C2 is executed thus resulting in *control dependence*.

Dependence among classes and components has an impact on the sequence in which they are integrated and tested. In the remainder of this section we define different types of dependencies among classes in an object-oriented application such as those written in Java, C#, or C++. The type of dependence among classes is used in Section 11.6 in the algorithms for determining a component integration sequence. The dependences mentioned here are applicable to a collection of classes as well as a collection of components each of which might be a collection of one or more classes.

Dependence among components affects the sequence in which they are combined and tested.

Figure 11.2 shows a graphical means of representing relationships among components. Such a graph is also known as *Object Relation Diagram*, or simply referred to as an ORD. We use the notation in Figure 11.2(a) or (b) when there is no need to specify the type of relationship and (c) when the specific relationship is of importance. Recall that a component could be as small as a class or as large as a subsystem. The algorithms presented in Section 11.6 are likely to be of practical use when the components to be integrated are fairly complex rather than simple classes. Nevertheless, these algorithms can be used regardless of the complexity of the components to be integrated.

An Object Relation Diagram (ORD) is a convenient structure to capture the dependence among components in a subsystem or a system.

While an ORD is a useful means to exhibit inter-component relations, it is certainly not the only means. Several details of component interactions are difficult or nearly almost impossible to describe in an ORD. In such cases, one uses textual means to describe the details of the interfaces. One such document, used by NASA when developing software systems for its spacecraft, is known as *Interface Control Document*, abbreviated as ICD. In the automobile industry, the AUTOSAR standard specified interfaces using Interface Files or the Software Specification (SWS) documents.

Figure 11.2 (a) Component C1 depends on component C2. (b) Components C1 and C2 depend on each other thus creating a cycle. (c) Component C1 depends on component C2 and the dependency type is `T`. Here `T` denotes some form of dependency described in Section 11.3.1.

ORDs can be used to express object relationships in an embedded system consisting of hardware and software.

Obtaining precise relationships among objects might not always be possible, and especially so when the complete code of the system under test is unavailable.

Example 11.2 Automobile engine control systems perform a number of time-dependent tasks. Often these tasks are performed by software components that are connected with each other over a network bus. As shown in Figure 11.3, a software component might receive data from one or more sensors and send control signals to an automobile component via one or more controllers.

In an environment like the one described above, each software component is treated as a black box. The actual code of the component might not be available though one assumes that the component itself has been well tested prior to integration testing. Thus, it might not be possible, or even feasible, to obtain a precise ORD for the components such as the one shown later in Figure 11.6. However, given the interface description, it should be possible to create a component dependence diagram that is similar to an ORD but that indicates dependence and not necessarily its type. Integration testing in such cases is based on precise interface requirements.

For example, it might be required that SW-C1 obtains data from a sensor, processes it within 100 milliseconds, and sends a suitable message to SW-C2. In turn, SW-C2 consults a table, generates, and sends another data item to SW-C3. This component in turn computes a signal value from the data received and sends it to a controller. In this case, we know the dependencies among the various software components, sensors, and controllers. Hence a component

Figure 11.3 A partial chain consisting of software components in the engine control management of an automobile engine control management. Three software components (prefixed as SW-C) are shown. The components communicate via a bus. They also gather data from sensors and send signals to controllers.

Figure 11.4 (a) Class C2 inherits from class C1. (b) Class C1 aggregates objects of class C2. (c) Class C1 uses data and or methods in class C2.

dependence diagram can be drawn and used to decide a suitable integration sequence.

11.3.1 Class relationships: static

Classes in an object-oriented program are often related to each other. In its most generic form, such a relation is referred to as an *association*. During integration testing, it is sometimes useful to consider three types of relations: inheritance, aggregation, and association. The aggregation relationship is also known as *composition*.

Inheritance, aggregation (composition), and association are three commonly used relationships among classes.

The inheritance relation indicates that a class C2 inherits data and or methods from another class C1. As in Figure 11.4(a), the inheritance relationship is indicated by drawing an arrow from C2 to C1 and labeling it as "I" meaning that C2 inherits from C1. For example, in Java, class C2 can be made to inherit from class C1 by writing the class header as follows:

Class relationships could derived using static analysis of program code. However, relationships thus derived are static and optimistic.

```
public class C2 extends C1{  //  Extension of C1
....
}
```

Figure 11.4(b) indicates that class C1 aggregates objects of class C2. Note that we label the arrow from C1 to C2 as "Ag." For example, a class named `Graph` denoting a graph may aggregate objects of type `Node`. Further, `Node` might aggregate objects of type `Edge`. As another example, a class named `Picture` might aggregate several objects of type `Point`. The following code segment indicates aggregation of C2 objects:

```
public class C1  {
C2[ ] C2Objects=new C2[100]; // Aggregation of C2 objects.
C2 aC2Object=new C2; // C1 uses a C2 object.
....
}
```

Figure 11.4(c) indicates that class C1 has an association with class C2 meaning thereby that C1 uses some data or method in C2. The association edge is labeled "As." The following code segment indicates three possible ways in which C1 can associate itself with C2:

An association relation between two classes could arise due to a method call, or data use, by one class into another class.

```
public class C1 {
...
int d=C2.getVal( ); // Use of method getVal( ) in C2.
float p=C2.x+1; // C1 accesses x in C2.
public void aMethod(C2 z); // C1 uses a field in C2.
....
}
```

The first of the above three statements indicates *method call* association as class `C1` uses a method in class `C2`. The second statement indicates *field access* association as class `C1` uses variable `x` defined in `C2`. The last statement is an example of *parameter* association as class `C1` uses a parameter of type `C2` in method `aMethod`. It may be worth noting that aggregation is also a kind of association. However, in integration testing it is sometimes useful to distinguish among different types of associations as explained above.

It is also possible for a class to have more than one type of dependence on another class. Consider, for example, the following statements:

```
public class C1  {
  public void method mC( ){ . . . }
    ...
}// End of C1
public class C2 extends C1 {
  public C2( C1 c){c.mC( );}
    ...
}// End of C2
```

Clearly, in the above code segment, in an ORD, `C2` has an edge to class `C1` labeled `I`. In addition, due to the parameter `c` in the constructor, `C2` also associates with `C1`. In such cases the edge from `C2` to `C1` can be labeled `I/As`.

11.3.2 Class relationships: dynamic

The relationships shown in an ORD can be derived at compile time. Hence, these are known as *static* relationships. However, when testing a collection of classes, it is important to know class dependencies that arise due to polymorphism or indirection. Such dependence of one class on the other is also

Figure 11.5 (a) Static dependence ORD. (b) Dynamic dependence ORD derived from (a).

known as *dynamic dependence*. The following example illustrates dynamic dependence.

Inter-class relationships that arise due to polymorphism or indirection are known as dynamic dependence relationships.

Example 11.3 Figure 11.5(a) shows the relationships among seven classes labeled `A` through `G`. Note that classes `C` and `D` are subclasses of class `B`. Class `A` associates with class `B` and hence, due to polymorphism, it also associates with classes `C`, `D`, and `G` as shown in Figure 11.5(b). Further, class `B` aggregates objects of class `F`. Hence, transitively, class `A` also depends on class `F`.

The ORD in Figure 11.5(b) suggests that if, for example, class `F` is changed, then class `B` as well as class `A` ought to be tested. Similarly, when any of classes `C`, `D`, and `G` is modified, `A` needs to be retested.

11.3.3 Class firewalls

A firewall for class `X` is the set of all classes that may be affected if `X` is modified. Thus, any class that depends on `X` will be in the class firewall of `X`. Firewalls can be determined from the ORD. Let F_X denote the firewall for class `X`. All classes in F_X are targets for retesting when `X` is modified. Thus, firewalls are useful in determining what to test during regression as well as integration testing.

All classes that depend on some class C are considered to be in the firewall of C.

It is easy to determine a class firewall from an ORD using the following *depends* relation D:

$D = \{(\mathtt{X}, \mathtt{Y}) | \text{when there is an edge in the ORD from class } \mathtt{Y} \text{ to class } \mathtt{X}\}$.

Let D^+ denote the irreflexive transitive closure of D defined as follows:

$D^+ = \{(\mathtt{X}, \mathtt{Z}) | (\mathtt{X}, \mathtt{Y}) \text{ and } (\mathtt{Y}, \mathtt{Z}) \in D\}$.

Using D we can define the firewall $F_{\mathtt{X}}$ for class X as follows:

$F_{\mathtt{X}} = \{\mathtt{Z} | (\mathtt{X}, \mathtt{Z}) \in D^+\}$

An ORD can be used to determine the firewall of each class using a "depends" relation.

Example 11.4 Let us determine the firewalls of all classes in Figure 11.5(a). Relations D and D^+ can be computed by an examination of the ORD as follows:

$D = \{(\mathtt{B}, \mathtt{A}), (\mathtt{E}, \mathtt{B}), (\mathtt{F}, \mathtt{B}), (\mathtt{B}, \mathtt{C}), (\mathtt{B}, \mathtt{D}), (\mathtt{D}, \mathtt{C}), (\mathtt{D}, \mathtt{G})\}$.
$D^+ = D \cup \{(\mathtt{B}, \mathtt{G}), (\mathtt{E}, \mathtt{C})(\mathtt{E}, \mathtt{D}), (\mathtt{E}, \mathtt{G}), (\mathtt{F}, \mathtt{C})(\mathtt{F}, \mathtt{D}), (\mathtt{F}, \mathtt{G})\}$.

Using D^+ as above, the firewalls for each class in Figure 11.5(a) can be computed as follows:

Class	Firewall	Class	Firewall
A	Ø	B	A, C, D, G
C	D	D	G
E	B, C, D, G	F	B, C, D, G
G	Ø		

11.3.4 Precise and imprecise relationships

An ORD derived from the code of an application is likely to be more precise than one derived from its design.

An ORD can be derived from design diagrams during the early stages of software development. For example, class diagrams in UML could be a source of an ORD for an application. However, such an ORD is likely to be *imprecise* in the sense that it might show relationships that do not actually exist in the application code. Of course, this imprecision could be despite the fact that the application conforms to the design. While such an imprecise ORD is useful in generating integration test plans and for advance test planning, it might indicate stubs that are not needed. Hence, an ORD generated from a design is best useful for early planning, but not necessarily for the determination of the actual integration order, or regression test, or for code coverage analysis.

Once the application code is available, it is possible using static analysis of the code to refine the ORD constructed using the design. In some cases, one might like to construct a new ORD from the code. The ORD created from the code will likely exhibit a more accurate set of inter-class relationships than the ORD created using only the design. Of course, keep in mind that sometimes one reason why stubs are created is because the corresponding code is not available. In such a situation, one needs to rely on a mix of the design diagrams and whatever code is available.

Figure 11.6 ORDs corresponding to Program P11.1. (a) Imprecise ORD. (b) Precise ORD.

Example 11.5 Consider a Java Program P11.1. It contains five classes. Class `T` is the top level class for testing the remaining classes. Class `Y` is a subclass of `X`. Class `Z` makes use of an object of type `X` in its constructor. Figure 11.6(a) shows an ORD assumed to be developed from the design diagram of this program. It shows the appropriate edges between the various classes.

Program P11.1

```
public class X{
  public void  print( ){
          System.out.println("This is from X.");
        }
}//  End of X
public class Y extends X{
        public void print( ){
              System.out.println("This is from Y.");
        }
  }//  End of Y
  public class Z{
          public Z (X xObject){
                xObject.print( );
          }
  }//  End of Z
  public class T{
          public static void main(String [ ] args){
                Z zObject1=new Z(new Y( ));
          }
  }//  End of T
```

An examination of Program P11.1 reveals that `T` creates an object of type `Z` by passing an object of type `Y`. Further, an examination of

the code of all classes reveals that Z will never actually use an object of type X. Thus, it is more appropriate to indicate an association relationship between Z and Y than to show a relationship between Z and X. This more precise relationship is shown in Figure 11.6(b).

The precise ORD in Figure 11.6(b) also shows exactly where the dependencies arise. For example, it shows that the dependence of T on Z is from the statement at line 18. Similarly, it also shows that the dependence of Z on Y is from the statement at line 12 in Z and the statement at line 18 in T.

Precise ORDs are helpful in several situations. First, they help determine exactly which classes ought to be stubbed. Second, they help determine which portion of the class code ought to be covered when integrating two classes. And thirdly, during regression testing they help determine which classes ought to be retested when there is a change in one or more classes.

11.4 OO versus Non-OO Programs

In programs written in non-OO languages such as C, the call graph or the program dependence graph across components can be used to capture relationships among various components

The class relationships mentioned above are obviously for object-oriented (OO) programs written in languages such as Java and C++. In non-OO languages such as C, one can use other types of dependence graphs. For example, a call graph can be used to capture the dependence among various methods in a program. Note that a call graph can also be constructed for OO-programs. In fact a precise association relation between two objects indicates, among others, the calling relationships among methods across various objects.

For the purpose of integration testing, the call graph can be constructed so that only calls across different components to be integrated are captured while those within a component are ignored. Program dependence graphs introduced in Chapter 1 can be used to capture more precise dependence relationships such as data and control dependence.

An ORD or a call graph can be used to derive a test order for integration testing.

Once the dependence among various components is captured, algorithms described in Section 11.6 can be used to help arrive at a test order. This order may simply be derived assuming an abstract dependence such as "component A depends on component B." More detailed dependence types could also be used, such as "component A calls a method in component B." While the abstract dependence is useful in deriving a test order, the more precise dependence is useful in deriving the tests and assessing their adequacy.

Figure 11.7 (a) A hypothetical set of classes and their relationships. (b) A sample top-down integration sequence. (c) A sample bottom-up integration sequence; drivers are needed for C5, C4, C2, and C3.

11.5 Integration Hierarchy

There are several strategies for integrating components prior to testing. Three strategies, namely, *big-bang*, *top-down*, and *bottom-up*, are explained next using the class hierarchy in Figure 11.7.

Components can be integrated in several ways prior to testing. The big bang approach integrates all components at once while the top down and bottom up strategies are incremental.

Big-bang integration: The big-bang strategy is to integrate all components in one step and test the entire system. Doing so does save time in writing stubs and drivers. When using this strategy for the system in Figure 11.7(a), all five classes shown will be integrated and tested together.

When using the big-bang strategy the identification of sources of failure, and correcting the errors, becomes increasingly more difficult with the complexity of the system. One may use this strategy when the number of components involved is small and each is relatively simple. Thus, big-bang integration might be useful when an individual programmer is testing self-designed and coded components of a large system.

The big-bang integration and testing is not the same as system testing.

It is important to note that big-bang integration testing is not the same as system testing.The goal in integration testing is to ensure that there are no integration errors, whereas in system testing one tests the entire system with several other objectives such as the correctness of the system level functionality, performance, and reliability.

The top-down approach integrates components starting from the top and moving down the class hierarchy. Doing so may require many stubs.

Top-down integration: As the name implies, a strictly followed top-down integration strategy requires that components be integrated from the top moving down in the class hierarchy. Thus, for the hierarchy in Figure 11.7(a), one would begin testing class C1, then integrate it with C2 or C3, and test the combination {C1, C2} or {C1, C3} as the case may be. In the next step, the set of classes {C1, C2, C3} are integrated and tested. Finally, the complete set of five classes will be tested. As shown in Figure 11.7(b), top-down testing requires the use of stubs. Thus, when testing C1 alone, one needs stubs for C2 and C3 because C1 depends on C2 and C3. Similarly when testing the combination {C1, C2, C3}, one needs stubs for class C4.

In the above, we have described a strict interpretation of top-down integration. In practice, the test team might decide to integrate in one of many different ways. For example, if classes C1, C2, and C3 are all available and have been individually tested, then they can be integrated and tested together. In the next step, assuming that classes C4 and C5 are available, the entire set of classes can be integrated and tested. Thus, instead of using the steps described earlier, one may perform the integration in only two steps. Toward the end of this section we examine the factors that help a test team decide what to integrate and when.

The bottom-up approach integrates components starting from the bottom of e class hierarchy and moving up. Doing so may require several drivers.

Bottom-up integration: In strict bottom-up integration one moves up the component hierarchy to integrate and test components. As shown in Figure 11.7(c), class C5 is tested first. Next, C1 is integrated with C4 and the combination tested. The integration proceeds until all components have been integrated and tested. Bottom-up testing requires the creation of drivers. For example, when testing C5, one needs a driver to make "use of" or to "drive" the execution of code in C5 that is used by classes that depend on it. Thus, whereas top-down strategy may require the creation of stubs, bottom-up strategy may require the creation of drivers. The number of stubs or drivers needed obviously depends on how the components are integrated.

As in top-down integration, there are several ways in which components can be integrated and tested. For example, assuming the hierarchy in Figure 11.7(a), one could integrate C4 and C5 and test them together. In the next step, {C2, C3, C4, C3} can be tested. Finally, the entire set of classes can be integrated and tested.

A mix of top-down and bottom-up strategies are likely to be used in testing large systems.

Sandwich strategy: This strategy combines the top-down and bottom-up strategies to decide the sequence of integration and testing. For example, the classes in Figure 11.7(a) could be integrated and tested in the following order: {C4, C5}, {C1, C2, C3}, and finally {C1, C2, C3, C4, C5}.

This sequence uses bottom-up integration when combining C4 and C5 and top-down integration when combining C1, C2, and C3.

11.5.1 Choosing an integration strategy

The choice of an integration strategy, and the integration sequence, depends on several factors, some of which are enumerated below.

Which integration strategy to adopt depends on several factors including code availability, the difficulty in the construction of stubs and drivers, the size and experience of the test team and the availability of tools.

1. *Availability of code*: With reference to Figure 11.7(a), suppose that classes C1 and C2 are available but the remaining are not. In this case one has the option of either delaying the test until, say, C3 is available, or integrate C1 and C2 and test this combination along with a stub for C3. Thus, code availability may determine the integration sequence.
2. *Difficulty or ease of constructing stubs*: In some cases the dependence of a class on another class might be so significant that the best way to test one or the other is to test both together. For example, consider four classes named `CreateGraph`, `Graph`, `Node`, and `Edge`. These classes are related to each other as shown in Figure 11.8. `CreateGraph` reads data on nodes and edges from a file and creates a `Graph` object. This object contains several node objects. Each `Node` object contains zero or more `Edge` objects.

   Testing the class `Graph` alone will require stubs for classes `Node` and `Edge`. Given that these three classes are tightly coupled, it might be best to test them together rather than in a top-down or bottom-up manner.

   Of course, while the example above is from a real program, the classes themselves might be considered simple enough and hence ought to be integrated together. However, scenarios such as the one above might occur in practice where a tester may decide to integrate and test in one step rather than create stubs and test in an incremental manner.
3. *Difficulty or ease of constructing drivers*: Consider again the four classes and their relationship in Figure 11.8. This example can be used to argue that it would be best to integrate and test the four classes in one step rather than create drivers for `Edge`, `Node`, and test them in a bottom-up manner. In this example, bottom-up testing would imply testing the `Edge` class first which in turn would require a driver to be written. However, `CreateGraph` already has the code that drives the methods in `Edge`. The same is also true for `Node` ad `Graph` for which `CreateGraph` has the driver code. Hence, in this example, big-bang rather than a bottom-up, seems to be the best strategy.
4. *Size of the test team*: Depending on the number of testers available one might be able to do some parallel integration rather than doing everything sequentially. Consider, for example, the classes in Figure 11.7.

Figure 11.8 Class `Graph` aggregates objects of type `Node` and class `Node` aggregates objects of type `Edge`. Class `CreateGraph` reads node and edge data from a file and creates an object of type `Graph`. "C" indicates the "aggregation" relation which is also known as the "composite," relation. "A" indicates the association relation.

In top-down testing, we could first test component C1, followed by {C1, C2}, followed by {C1, C3}, and so on. However, if two testers are available then {C1, C2} could be integrated and tested by one tester and, in parallel, {C1, C3} could be integrated and tested by another. Once both testers have completed their tasks, C1, C2, and C3 could be integrated and tested assuming that no integration errors were discovered in the previous step that might otherwise would prevent further integration.

11.5.2 Comparing integration strategies

Each of the integration strategies discussed above has their pros and cons as examined in the following:

Top-down integration requires the creation of stubs while bottom-up integration requires the creation of drivers.

1. *Stubs and drivers*: Other than the big-bang strategy, all require the creation of either stubs or drivers, or both. Thus, top-down integration requires the creation of stubs for components at the lower level. On the other hand the bottom-up integration strategy requires the creation of drivers. As discussed earlier, design of both stubs and drivers could be challenging and costly. Thus, neither strategy seems to have any advantage over the other when viewed from the point of view of stub or driver creation. Certainly, depending on the specific project, it might be easier to create stubs than drivers, or vice versa. In that case one strategy might be preferred over the other.
2. *Partially working program*: There are situations when the management wishes to demonstrate a system to a select group of potential customers. Top-down approach has the advantage that one could have a partially working system, also known as a *skeleton system*, before integration

testing is complete. Bottom-up approach builds the system from the bottom and hence a working system is not available until the top level components have been integrated. In most modern systems that use a graphical user interface (GUI), it might be best to begin integration into the GUI. Doing so would ensure that a skeletal system is always available regardless of whether the top-down or the bottom-up approach is used.

3. *Error location*: Prior to integration testing, one usually would not know the location of integration errors. However, based on past experience and skills of the development teams, as well as various software metrics, one might be able to flag components likely to be more erroneous than others.

11.5.3 Specific stubs and retesting

As mentioned earlier, a stub for class `C2` is needed when class `C1` depends on `C2` and `C2` is not available when testing `C1`. A stub is a temporary replacement for a class and hence it should be easy to write and test a stub. Obviously, this implies that a stub will perform simple tasks like always returning a fixed or a random data when requested rather than actually executing code to generate data as per the requirements. This simplicity might not allow for a thorough testing of `C1`. In any case, stubbing a class has an associated cost. Hence the cost ought to be a consideration when deciding which classes to stub when there is a choice.

Stubbing a class has an associate cost. This is one factor that influences decisions regarding what classes to stub.

When `C1` depends on class `C2` we say that `C1` is a *client* of `C2`. In this case `C2` is considered as a server class. Of course, it is possible that `C1` and `C2` depend on each other in which case both serve as client and server classes for each other. The stub for a server class depends on the requirements placed by its client. Thus, two stubs might be required for a single server class if it serves two clients. Such stubs are known as *specific* stubs. Each specific stub is written to simulate a simplified behavior of a server class so as to fulfill the testing needs of the corresponding client class.

Two stubs might be required for a class that serves two clients. Each such stub is known as a "specific stub."

The client class needs to be retested when the code for a stubbed class is tested. However, this retesting is limited to the edges that connect the client to the stubbed class. For example, if the only dependence between `C1` and `C2` is a method `m` in server class `C2` that is called by `C1`, then retesting of `C1` needs to focus only on the calls to `m`. Doing so reduces the test cases that are rerun during a retest. Nevertheless, retesting has an associated cost. Thus, when deciding which classes to stub, or not to stub, one needs to consider the cost of retesting. In cases where the dependence between two classes is significant, and there exists a cycle involving the two, it might be best to test the two together rather than test them incrementally.

Example 11.6 Figure 11.9(a) shows two client classes `C1` and `C2` that depend on server class `S`. If `S` is not available while testing `C1` or `C2`, a stub for `S` is needed as shown in Figure 11.9(b). However, to keep the stub simple, one may decide to create two specific stubs `S1` and `S2` as shown in Figure 11.9(c). In either case, when class `S` is available, `C1` needs to be retested with respect to edge `e1`. Similarly, class `C2` needs to be tested with respect to edge `e2`.

An ORD might have a cycle. In this situation at least one stub is needed if pure incremental testing is preferred.

Example 11.7 Consider the ORD in Figure 11.10. Note that this ORD contains a cycle between nodes `B` and `H`. Hence at least one stub is needed for incremental testing. Now consider the following test sequence: `A`, `E`, `C`, `F` using STUB(`F`, `D`), `H` using STUB(`H`, `B`), `D`, `B`, and `G`. In this sequence two specific stubs are needed as there is one client of each server class `D` and `F` needed by their respective client classes `F` and `H`. Now suppose it is decided to minimize the number of specific stubs. This could be done by using only one stub

Figure 11.9 (a) Client classes `C1` and `C2` and server class `S`. (b) Server classes `S` stubbed. (c) Specific stubs `S1` and `S2` for server class `S`.

Figure 11.10 A sample ORD with eight classes containing cycles between classes `B`, `H`.

class B and the following test sequence: A, E, C, H using STUB(H, B), D, F, B, G.
In the above test sequences the number of specific stubs is the same as the number of stubs. However, this is not always true as is shown later when we look at the algorithms for generating a test sequence. Also see Exercise 11.8.

The cost of retesting and stubbing is class and project dependent and hence it is difficult to offer any specific rules to decide whether or not to stub a class when dependence exists. The algorithms that follow attempt to minimize the number of stubs and the associated costs of stubbing and retesting.

The TD and the TJJM methods that follow focus on minimizing the stub count while the BLW method aims to minimize the cost of retesting and stubbing of server classes. Note that none of these methods claims to generate an optimal set of stubs. Also, as mentioned above, minimizing the number of stubs for incremental testing does not necessarily mean minimizing the cost of integration testing. Thus, even though the three methods described below are useful in practice, the outcome might need to be subjected to further project and code dependent considerations before the final decision is made by the test team on what or what not to stub.

Methods for determining near-optimal test order for classes aim to minimize the stub count and/or the cost of retesting.

11.6 Finding a Near-Optimal Test Order

The problem of finding an optimal integration sequence is NP-complete. Thus, several methods have been proposed to find a near-optimal integration sequence. Three such methods are introduced in the following sections. These are referred to as the Tai-Daniels (TD), Traon-Jéron-Jézéquel-Morel (TJJM), and Briand-Labiche-Wang (BLW) methods. Each of the methods described below takes an ORD as input in the form of a directed graph. It is assumed that each edge in the ORD is labeled by exactly one relationship, i.e. I, Ag, and As. Thus, in the event a class Y is associated with class X by both the I and As relations, it is best to use the As relation in the input ORD. A comparison of the three methods is given after each has been examined.

The problem of finding an optimal test order from a given ORD is NP complete.

11.6.1 The TD method

The TD method takes an ORD *G* as input and applies the following sequence of steps to generate an integration sequence.

Step A *Assign major level numbers 1, 2, 3, and so on to each node in G.* The assignment of major levels partitions the nodes in the ORD

into n categories, where $n \geq 1$ is the number of major levels. Nodes at major level 1 do not have any outgoing inheritance or aggregation edges. Nodes at major level 2 have inheritance or aggregation edges to nodes at level 1. In general, each node at major level i, $1 < i \leq n$, has an inheritance or an aggregation edge to nodes at major level $j < i$ and at least one such edge to a node at major level $(i - 1)$. Once partitioned in this way, the classes at major level i are considered for testing prior to those at major levels k, $k > i$. The order in which classes at major level i are to be tested is determined in the next step. A procedure named `assignMajorLevels` for assigning major level to each node in G discussed in the next page.

Step B *Assign minor level numbers 1, 2, 3, and so on to each node in G.* Note from the previous step that each major level has one or more nodes from G. A minor level is assigned to each node to decide the sequence in which the nodes at a major level are to be tested. Procedure `assignMinorLevel`. This procedure aims at assigning minor levels to nodes with the goal of minimizing the stubs needed for integration testing.

Step C *Generate test sequence, stubs, and retest edges.* Once the major and minor levels have been assigned, each node has two integers (i, j) associated with it, where i denotes its major level and j its minor level. Testing begins with nodes at major level 1. Among the nodes at major level 1, nodes at minor level 1 are tested first, followed by the nodes at minor level 2, and so on. Nodes at a major level with the same minor levels can be tested concurrently. Once the nodes at major level 1 have been tested, nodes at major level 2 are considered for testing, and so on. Note that testing a node A at major level i requires a stub for a node B that has an incoming association edge from A and is at a higher level, major or minor, than A. Also, while testing a node A, any edge to a node at a lower level, major or minor, needs to to be retested.

The following subsections describe the procedures corresponding to the three steps mentioned above. However, prior to going through the procedures mentioned below, you may view Figure 11.12 in case you are curious to know how an ORD looks like after its nodes have been assigned major and minor levels.

A. Assign major level numbers

Each node in an ORD G is assigned a major level number. A level number is an integer equal to or greater than 1. The purpose of assigning level

numbers will be clear when we discuss Step 3 of the method. The assignment of major level numbers to each node in G is carried out using a depth first search while ignoring the association edges to avoid cycles.

The Tai-Daniels method assigns major- and minor-level numbers to each node in an ORD. This assignment is done using depth-first search of the ORD.

The algorithm for assigning major level numbers as described next consists of two procedures. Procedure *assignMajorLevel* marks all nodes in the input ORD as "not visited." Then, for each node that is not visited yet it calls procedure *levelAssign*. The *levelAssign* procedure does a depth first search in the ORD starting at the input node. A node that has no successors is assigned a suitable major level number starting with 1. If node n_1 is the immediate ancestor of node n_2 then n_1 will be assigned a level one greater than the level of n_2.

`Procedure for assigning major level numbers.`

Input: An ORD G.
Output: ORD G with major level number (≥ 1) assigned to each node.

Procedure: `assignMajorLevel`

/* N denotes the set of all nodes in G. The major level number of each node is initially set to 0. *maxMajorLevel* denotes the maximum major level number assigned to any node and is initially set to 0.
The association edges are ignored when applying the steps below.
*/

Step 1 Mark each node in N as "not visited."
Step 2 Set `level = major`.
Step 3 For each node $n \in N$ marked as "not visited" apply the `levelAssign` procedure. A node that is visited has already been assigned a major level number other than 0.
Step 4 Return G.

`End of Procedure assignMajorLevel`

`Procedure for assigning a level number (major or minor) to a node.`

Input: (a) An ORD G. (b) A node $n \in G$. (c) `level`.
Output: G with major or minor level number assigned to node n depending on whether `level` is set to `major` or `minor`, respectively.
Procedure: `levelAssign`

/* This procedure assigns a major or minor level number to the input node n. It does so using a recursive depth first search in G starting at node n.
*/

Step 1 Mark n as "visited."

Step 2 Let S_n = successor nodes of n.

Step 3 `if` ($S_n = \emptyset$) `then` assign 1 to n as its major or minor level depending on `level` and return.

Step 4 `if` ($S \neq \emptyset$) `then` for each node $m \in S$ do the following:

4.1 `if` (m is not visited) `then` apply `levelAssign` to m.

4.2 Set `maxLevel` = maximum major (minor) level among all nodes in $S_n + 1$.

4.3 `if` (`maxMajorMinorLevel` < `maxLevel`) `then` `maxMajorMinorLevel=maxLevel`.

Step 5 Return G.

`End of Procedure levelAssign`

Example 11.8 Let us now apply procedure `assignMajorLevel` to the graph G in Figure 11.11(b) and assign major levels to each node in the graph. The set of nodes $N = \{A, B, C, D, E, F, G, H\}$. In accordance with Step 1, begin by marking all nodes in N to "not visited." As no node has been assigned a major level, set *maxMajorLevel=0*. A node is indicated as "not visited" by adding a superscript n to its name. Thus, A^n indicates that node A is not visited. Also, the successors of node X are denoted as S_X.

Figure 11.11 (a) An ORD consisting of nodes (classes) labeled A through H. This ORD contains three inheritance edges labeled I, 11 association edges labeled A, and three aggregation edges labeled Ag. (b) ORD in (a) without the association edges. This is used to illustrate the TD method.

Now move to Step 3. Let us start with node A^n. As this node is not visited, enter procedure `levelAssign`. The ensuing steps are described below.

`levelAssign`.Step 1: Mark node A^n as visited.

`levelAssign`.Step 2: As node A has no successors, $S_A = \emptyset$.

`levelAssign`.Step 3: Assign major level 1 to A and return from `levelAssign`.

`assignMajorLevel`.Step 3: Select the next node in N. Let this be node B^n.

`levelAssign`.Step 1: Mark B^n as visited.

`levelAssign`.Step 2: Get the successor set for node B. Thus, $S_B = \{H^n\}$.

`levelAssign`.Step 4: Select the next node in S_B, which is H^n.

`levelAssign`.Step 1: This is a recursive step. Mark H^n to visited.

`levelAssign`.Step 2: Get the successors of H. $S_H = \{C^n\}$.

`levelAssign`.Step 4: Select the next node in S_H, which is C^n.

`levelAssign`.Step 1: As C^n is not visited we apply `levelAssign` to this node.

`levelAssign`.Step 1: Mark node C^n as visited.

`levelAssign`.Step 2: As node C has no successors, $S = \emptyset$.

`levelAssign`.Step 3: Assign major level 1 to C and return from `levelAssign`.

`levelAssign`.Step 2: Set `level=maxMajorLevel+1=2`.

`levelAssign`.Step 3: Assign major level 2 to H. Return from `levelAssign`.

The remaining steps are left to the reader as an exercise. At the end of procedure `assignMajorLevel` the following major level numbers are assigned to the nodes in G.

Major level 1 assigned to nodes A, E, and C.

Major level 2 assigned to nodes F and H.

Figure 11.12 Graph *G* from Figure 11.11(a) with major and minor levels assigned. Nodes at the same major levels are placed just above the corresponding horizontal line. Minor level numbers are placed just above each node. Edge deletion is explained in Example 11.9.

Major level 3 assigned to nodes *D* and *B*.

Major level 4 assigned to node *G*.

Figure 11.12 shows graph *G* with the major levels assigned.

B. Assign minor level numbers

Nodes at the same major level are assigned minor level numbers.

In this step minor level numbers are assigned to each node. Assignment of minor level numbers is done for nodes within each major level. Thus, nodes at major level 1 are processed first, followed by nodes at the next higher major level, and so on. Of course, the processing of nodes at different major levels is independent and hence which set of nodes is assigned minor level numbers does not in any way affect the resulting assignment. The procedure for assigning minor level numbers follows.

Procedure for assigning minor level numbers.

Inputs: (a) An ORD *G* with minor level numbers assigned to each node.
(b) `maxMajorLevel` that denotes the maximum major level assigned by Procedure `assignMajorLevel`.

Output: ORD *G* with minor level numbers assigned to each node in *G*.

Procedure: `assignMinorLevel`
Perform the following steps for each major level i, $1 \leq i \leq$ `maxMajorLevel`.

Step 1 Let N_i be the set of all nodes at major level i. `if` ($|N_i| = 1$) `then` assign 1 as the minor level to the only node in N_i `else` move to the next step.
Step 2 `if` (nodes in N_i do not form any cycle) `then` assign minor level numbers to the nodes in N_i using procedure `assignMinorLevelNumbers`.
Step 3 Break cycles by applying procedure `breakCycles`.
Step 4 Use procedure `assignMinorLevelNumbers` to assign minor level numbers to the nodes in N_i.
Step 5 Return G.

End of Procedure `assignMinorLevel`

At step 2 in Procedure `assignMinorLevel` we need to determine all strongly connected components in G'. This can be done by applying, for example, Tarjan's algorithm to the nodes at a major level. The strongly connected components are then used to determine whether or not a cycle exists. Tarjan's algorithm is not described in this book but see the Bibliographic Notes section for a reference.

Assignment of minor levels requires the determination of strongly connected components in a ORD. One could use Tarjan's algorithm to find strongly connected components in an ORD.

Procedure `breakCycles` is called when there exist one or more cycles among the nodes at a major level. Recall that cycles among a set of nodes might be formed due to the existence of association edges among them. The `breakCycles` procedure takes a set of nodes that form a cycle as input and breaks all cycles by suitably selecting and removing association edges. Cycles are first broken by removing edges ensuring that a minimum number of stubs is needed. However, doing so might not break all cycles. Thus, if cycles remain, additional association edges are removed using *edge weight* as the criterion as explained in the procedure below.

A special procedure is needed when there exist cycles among the nodes at a major level. Cycles are broken with the aim of minimizing the stubs needed.

Procedure for breaking cycles in a subgraph of an ORD.
Inputs: (a) A subgraph G' of ORD G that contains one or more cycles and of which all nodes are at the same major level.
(b) Integer m, where $1 \leq m \leq maxMajorLevel$, is the major level of the nodes in G'.
Output: Modified G' obtained by the removal of zero or more edges from the input subgraph.
Procedure: `breakCycles`

Step 1 Let $X \in G'$ be a node such that there is an association edge to X from a node at a major level lower than m. Let S denote the set of all such nodes. Note that if a node X in G' has an association edge

coming into it from a node at a lower major level then there already exists a stub for X because all nodes at major level less than m are tested prior to testing the nodes at level m.

Step 2 `if` (S is non-empty) `then` construct E as the set of all association edges from nodes in G' that are incident at nodes in S `else` go to Step 5.

Step 3 `if` (E is non-empty) `then` remove all edges in E from G'.

Step 4 `if` (G' contains no cycles) `then` return `else` move to the next step.

Step 5 Compute the weight of each edge in G' that has not been deleted. Let $n1$ be the source and $n2$ the destination node of an edge e. Then the weight of e is $s+d$, where s is the number of edges incident upon $n1$ and d is the number of outgoing edges from $n2$.

Step 6 Let $Es = \{e1, e2, \ldots, en\}$ be a list of all undeleted edges in G' sorted in the decreasing order of their weights. Thus, the weight of $e1$ is greater than or equal to that of $e2$, that of $e2$ is greater than or equal to the weight of $e3$, and so on. Delete from G' the edges in Es one by one starting with $e1$. Continue the process until no cycles remain in G'.

Step 7 Return G'.

End of Procedure `breakCycles`

The next procedure assigns minor level numbers to all nodes at a major level given that they do not form a cycle. This procedure is similar to procedure `assignMajorLevel`. It takes a subgraph G' of the original ORD G as input. G' has no cycles. It begins by marking all nodes in G' to "not visited." It then sets `level` to `minor` and calls procedure `levelAssign` to assign the minor level numbers to each node in G'.

Procedure for assigning minor level numbers to nodes at a major level.

Input: Subgraph G' of G with no cycles.
Output: G' with minor level numbers assigned to each node.

Procedure: `assignMinorLevelNumbers`

Step 1 Mark each node in G' as "not visited."

Step 2 Set `level=minor`.

Step 3 For each node $n \in G'$ marked as "not visited" apply the `levelAssign` procedure. A node that is visited has already been assigned a minor level number other than 0.

End of Procedure assignMinorLevelNumbers

Example 11.9 Let us now assign minor level numbers to the ORD G in Figure 11.11(a). The major level numbers have already been assigned to the nodes and are shown in Figure 11.12. Begin by applying Procedure `assignMinorLevel`.

`assignMinorLevel`. Let $i = 1$. Thus, the first set of nodes to be processed are at major level 1.

`assignMinorLevel`.Step 1: $N_1 = \{A, C, E\}$. As N_1 has more than one node, we move to the next step.

`assignMinorLevel`.Step 2: As can be observed from Figure 11.12, N_1 has one strongly connected component consisting of nodes A, C, and E. This component has a cycle. Hence move to the next step with the subgraph of G' of G consisting of the nodes in N_1, and major level $m = 1$.

`breakCycles`.Step 1: As all nodes in G' are at major level 1, we get $S = \emptyset$.

`breakCycles`.Step 2: S is empty hence move to Step 5.

`breakCycles`.Step 5: The weight of each of the four edges in G' is computed as follows: A−C=2+2=4; C−E=1+1=2; E−A=1+1=2; and $\mathrm{C} - \mathrm{A} = 1 + 1 = 2$.

`breakCycles`.Step 6: $Es = \{\mathrm{A} - \mathrm{C}, \mathrm{C} - \mathrm{E}, \mathrm{E} - \mathrm{A}, \mathrm{C} - \mathrm{A}\}$. The first edge to be deleted is A−C as it has the highest weight. As should be evident from Figure 11.12 doing so removes the cycles in G'. Hence there is no need to delete additional edges.

`assignMinorLevel`.Step 4: Now that the only cycle in G' is removed, the minor level numbers can be assigned.

`assignMinorLevelNumbers`.Step 1: Mark all nodes in G' as "not visited."

`assignMinorLevelNumbers`.Step 2: `level=minor`.

`assignMinorLevelNumbers`.Step 3: We need to loop through all nodes in G' that are marked as "not visited." Let us select node A.

`levelAssign`.Step 1: Node A^n is marked as "visited."

`levelAssign`.Step 2: As the edge A−C has been deleted, A has no successors. Thus, $S = \emptyset$.

`levelAssign`.Step 3: Assign 1 as the minor level to node A.

The remainder of this example is left to the reader as an exercise. Moving forward, nodes E and C are assigned, respectively, minor levels 2 and 3. Next, nodes at major level 2 are processed as described above. This is followed by nodes at major levels 3 and 4. Note that there are no cycles among the strongly connected components at major levels 2, 3, and 4 and hence no edges need to be deleted. Figure 11.12 shows the final assignment of major and minor levels to all nodes in ORD G. Also indicated in this figure is the edge A−C deleted by using the edge–weight criterion.

C. Generate test sequence, stubs, and retest edges

The test and retest sequence, stubs and retesting edges are determined once the major and minor levels have been assigned.

Let us now look into how the test sequence, stubs, and edges to be retested are generated. As mentioned earlier, the basic idea is to test all nodes at major level 1, followed by those at major level 2, and so on. Doing so may require one or more stubs when a node at major level i has dependency on a node at a major or minor level j, $i \leq j$. Procedure `generateTestSequence` generates the test sequence from an ORD G in which each node has been assigned a major and a minor level. The procedure also generates a list of nodes that ought to be stubbed and a set of edges that need to be retested.

Procedure for generating test and retest sequence.

Inputs: (a) ORD G with major and minor level numbers assigned.
(b) `maxMajorLevel`.
(c) `maxMinorLeve[i]` for each major level $1 \leq i \leq$ `maxMajorLevel`.

Outputs: (a) TestSequence, (b) stubs, and (c) edges to be retested.

Procedure: `generateTestSequence`

Step 1 Let `testSeq`, `stubs`, and `retest` denote, respectively, the sequence in which nodes in G are to be tested, a list of stubs, and a list of edges to be retested. Initialize `testSeq`, `stubs`, and `retest` to empty.

Step 2 Execute the following steps until Step 2.1.9 for $1 \leq i \leq$ `maxMajorLevel`.

2.1 Execute the following steps until Step 2.1.9 for minor level j, $1 \leq j \leq$ `maxMinorLevel[i]`.

2.1.1 Let `IT` be the set of nodes at major level i and minor level j that are connected by an association edge. Note that these nodes are to be tested together. If there are no such nodes then `IT`= $\{n\}$, where n is the sole node at level(i,j).

2.1.2 Update test sequence generated so far. `testSequence= testSequence+IT`.

2.1.3 Let N be the set of nodes at major level i and minor level j.

2.1.4 For each node $n \in N$, let $\texttt{SMajor}_n$ be the set of nodes in G at major level greater than i, $i <$ `maxMajorLevel`, such that there is an association edge from n to each node in `SMajor`. Such an edge implies the dependence of n on each node in `SMajor`. $\texttt{SMajor}_n = \emptyset$ for $i =$ `maxMajorLevel`.

2.1.5 For each node $n \in N$, let $\texttt{SMinor}_n$ be the set of nodes in G at minor level greater than j, $j <$ `maxMinorLevel[i]`, such that there is an association edge from n to each node in `SMinor`. $\texttt{SMinor}_n = \emptyset$ for $j =$ `maxMinorLevel[i]`.

2.1.6 Update the list of stubs generated so far. `stubs= stubs+`$\texttt{SMajor}_n$ `+`$\texttt{SMinor}_n$ for each $n \in N$.

2.1.7 For each node $n \in N$, let $\texttt{RMajor}_n$ be the set of nodes in G at major level less than i, $i <$ `maxMajorLevel`, such that there is an edge to n from a node in `RMajor`. $\texttt{RetestMajor}_n = \emptyset$ for $i =$ `maxMajorLevel`. Let `REMajor` be the set of edges to node n from each node in `RMajor`.

2.1.8 For each node $n \in N$, let $\texttt{RMinor}_n$ be the set of nodes in G at minor level less than j, $j <$ `maxMinorLevel[i]`, such that there is an edge to n from a node in `RMinor`. $\texttt{RMinor}_n = \emptyset$ for $j =$ `maxMinorLevel[i]`. Let `REMinor` be the set of edges to node n from each node in `Minor`.

2.1.9 Update the list of retest edges. `retest= retest+` $\texttt{REMajor}_n$`+` $\texttt{REMinor}_n$ for each $n \in N$.

Step 3 Return `testSequence`, `stubs`, and `retest`.

End of Procedure generateTestSequence

Example 11.10 Let us now apply Procedure `generateTestSequence` to the ORD in Figure 11.12.

`genTestSequence`.Step 1: `testSeq`=∅, `stubs`=∅, and `retest`=∅.

`genTestSequence`.Step 2: Set next major level $i = 1$.

`genTestSequence`.Step 2.1: Set next minor level $j = 1$.

`genTestSequence`.Step 2.1.1: Get nodes at major level 1 and minor level 1. `IT` $= \{A\}$.

`genTestSequence`.Step 2.1.2: `testSeq` $= \{A\}$.

`genTestSequence`.Step 2.1.3: $N = \{A\}$. A is the only node at major level 1 and minor level 1.

`genTestSequence`.Step 2.1.4: $\texttt{SMajor}_C$.

`genTestSequence`.Step 2.1.5: `SMinor` $= \emptyset$.

`genTestSequence`.Step 2.1.6: `stubs` $= \{C\}$.

`genTestSequence`.Step 2.1.7: $\texttt{RMajor}_A = \{\ \}$. $\texttt{REMajor}_A = \{\ \}$.

`genTestSequence`.Step 2.1.8: $\texttt{RMinor}_A = \{\ \}$. $\texttt{REMinor}_A = \{\ \}$.

`genTestSequence`.Step 2.1.9: No change in `retest`.

`genTestSequence`.Step 2.1: Set next minor level $j = 2$.

`genTestSequence`.Step 2.1.1: Get nodes at major level 1 and minor level 2. `IT` $= \{E\}$.

`genTestSequence`.Step 2.1.2: `testSeq` $= \{A, E\}$.

`genTestSequence`.Step 2.1.3: $N = \{E\}$.

`genTestSequence`.Step 2.1.4: $\texttt{SMajor}_E = \{F\}$.

`genTestSequence`.Step 2.1.5: $\texttt{SMinor}_E = \emptyset$.

`genTestSequence`.Step 2.1.6: `stubs` $= \{C, F\}$.

`genTestSequence`.Step 2.1.7: $\texttt{RMajor}_E = \emptyset$. $\texttt{REMajor}_E = \emptyset$.

`genTestSequence`.Step 2.1.8: $\texttt{RMinor}_E = \emptyset$. $\texttt{REMinor}_E = \emptyset$.

`genTestSequence`.Step 2.1.9: No change in `retest`.

`genTestSequence`.Step 2.1: Set next minor level $j = 3$.

`genTestSequence`.Step 2.1.1: Get nodes at major level 1 and minor level 3. `IT` $= \{C\}$.

`genTestSequence`.Step 2.1.2: `testSeq` $= \{A, E, C\}$.

`genTestSequence`.Step 2.1.3: $N = \{C\}$. C is the only node at major level 1 and minor level 1.

`genTestSequence`.Step 2.1.4: $\texttt{SMajor}_C = \{H\}$.

`genTestSequence`.Step 2.1.5: $\texttt{SMinor}_C = \emptyset$.

`genTestSequence`.Step 2.1.6: `stubs` $= \{C, F, H\}$.

`genTestSequence`.Step 2.1.7: $\texttt{RMajor}_C = \emptyset$. $\texttt{REMajor}_C = \emptyset$.

`genTestSequence`.Step 2.1.8: $\texttt{RMinor}_C = \emptyset$. $\texttt{REMinor}_C = \{\text{A}-\text{C}\}$.

`genTestSequence`.Step 2.1.9: `retest` $= \{\text{A}-\text{C}\}$. This implies that when testing class C, the association edge A−C needs to be retested because A was tested with a stub of C and not the class C itself.

The remainder of this example is left as an exercise for the reader to complete. The final class test sequence is as follows:

$A \rightarrow E \rightarrow C \rightarrow F \rightarrow H \rightarrow D \rightarrow B \rightarrow G$

The following stubs are needed:

Stub for C used by A

Stub for F used by E

Stub for H used by C

Stub for D used by F

Stub for B used by H

The following edges must be retested:

When testing C retest A−C.

When testing F retest E−F.

When testing H retest C−H.

When testing D retest F−D.

When testing B retest H−B.

11.6.2 The TJJM method

The TJJM method uses a significantly different approach than the TD algorithm for the determination of stubs. First, it does not distinguish between the types of dependance. Thus, the edges in the ORD input to the TJJM algorithm are not labeled. Second, it uses Tarjan's algorithm recursively to break cycles until no cycles remain. Third, while breaking a cycle, the weight associated with each node is computed differently as described below. Following are the key steps used in the TJJM algorithm starting with an ORD G.

The TJJM method does not make use of the edge labels in the given ORD. It does use Tarjan's algorithm recursively to break cycles until no cycles remain.

Step A Find the strongly connect components (SCCs) in G. Suppose $n > 0$ SCCs are obtained. Each node in an SCC has an index associated with it indicating the sequence in which it was traversed while traversing the nodes in G to find the SCCs. An SCC with no cycles is considered trivial. Only non-trivial SCCs are considered in the following steps. Tarjan's algorithm can be applied to find the SCCs in G.

Step B For each non-trivial SCC s found in the previous step, and for each node k in s, identify edges that are coming into k from its descendants within s. A node m is a descendant of node k if the traversal index of m is greater than that of node k implying that node m was traversed after node k. All such edges are known as *frond* edges.

Step C For each non-trivial SCC, compute the weight of each node as the sum of the number of incoming and outgoing frond edges.

Step D In each non-trivial SCC select the node with the highest weight as the root of that SCC. Remove all edges coming into the selected root. The selected node is also marked as a stub.

Step E In this recursive step, for each non-trivial SCC s, set $G = s$ and apply this method starting from Step A.

The above algorithm terminates when all SCCs to be processed in the steps following Step A are non-trivial. Upon termination of the algorithm one obtains one or more directed acyclic graphs consisting of nodes (classes) in G. Topological sort is now applied to each DAG to obtain the test order for the classes.

Procedure for generating stubs and test order.

Inputs: (a) ORD G and (b) `Stubs`.
Outputs: (a) `Stubs` and (b) Modified G with no cycles.

Procedure: `TJJM`

A non-trivial strongly connected component is one with no cycles.

Step 1 Find the set S of non-trivial strongly connected components in G. A non-trivial component is one with no cycles. Each node in G is assigned a unique traversal index. This index is an integer, starting at 1, that denotes the sequence in which the node was traversed while finding the strongly connected components. For any two nodes n and m in a component $s \in S$, n is considered an ancestor of m if its traversal index is smaller than that of m.

Figure 11.13 Strongly connected components found by applying the TJJM algorithm to the ORD in Figure 11.11. Nodes with two circles are selected roots of an SCC. Each SCC is enclosed inside a rectangle and labeled as Sx (y), where x denotes the SCC number, and when indicated, y denotes the node selected as a root for recursive application to break further cycles. Traversal sequence index is placed against the corresponding node.

Step 2 Return if S is empty. Otherwise, for each component $s \in S$, do the following:

2.1 For each node $n \in s$ identify edges entering n from its descendants in s. Such edges are known as a *frond* edges.

An edge entering into a node from one of its descendents in a non-trivial strongly connected component is known as a "frond" edge

2.2 For each node $n \in s$ compute the weight of n as the sum of the number of incoming and outgoing frond edges.

2.3 Let $m \in s$ be the node with the highest weight. `Stubs` = `Stubs` $\cup \{m\}$. In case there are multiple nodes with the same highest weight, select one arbitrarily. Delete all edges entering m from within s. This step breaks cycles in s. However, some cycles may remain.

2.4 Set $G = s$ and apply procedure TJJM. Note that this is a recursive step.

`End of Procedure TJJM`

Example 11.11 Let us apply the above mentioned steps to the ORD in Figure 11.11. This ORD has 8 nodes and 17 edges.

Step 1 The first application of Tarjan's algorithm as in Step A leads to two SCCs labeled S1 and S2 and enclosed inside a rounded rectangle in Figure 11.13. SCC S1 is trivial as it contains only one node. SCC S2 contains the remaining seven nodes and has several cycles. The traversal index of each node is placed next to the node label. For example, the traversal index of node `F` is 1 and that of node `B` is 6. Thus we have $S = \{S2\}$.

Step 2 Set $s = S2$ and move to the next step.

Step 2.1 Frond edges are shown as dashed lines in Figure 11.13. For example, node `C` has two incoming and two outgoing frond edges. The incoming frond edges are from nodes `B` and `H` as these two nodes are descendants of `C`. The outgoing frond edges are to nodes `A` and `E` as both these nodes reancestors of node `C`.

Step 2.2 Using the frond edges shown in Figure 11.13, we obtain the following weights: `F`: 1; `A`, `B` and `E`: 2; `H`: 3; and `C`: 4.

Step 2.3 Node `C` has the highest weight and hence is selected as the root of SCC S2. This node has three incoming edges A−C, B−C and H−C which are deleted as shown in Figure 11.13. `Stubs` = `Stubs` ∪ {`C`}.

Step 2.4 Set $G = S2$ and execute procedure `TJJM`.

Step 1 Four SCCs are obtained labeled in Figure 11.13 as S2.1, S2.2, S2.3, and S2.4. Of these, S2.1 and S2.4 are trivial and hence not considered further. Thus, $S = \{S2.2, S2.3\}$.

Step 2 Set $s = S2.2$.

Step 2.1 As in Figure 11.13, edges B−H and D are the two fronds s.

Step 2.2 The weights of various nodes are: nodes `B` and `D`: 1 and node `H`: 2.

Step 2.3 Node `H` is selected as the root of S2.2. `Stubs` = `Stubs` ∪ {`H`}. Edges B−H and D−H are deleted as these are incident on node `H`.

Step 2.4 Set $G = S2.2$ and execute procedure `TJJM`.

Step 1 There are three SCCs in S2.2. Each of these is trivial and consists of, respectively, nodes `D`, `B`, and `H`. Thus, $S = \emptyset$. Return from this execution of `TJJM`.

Step 2 Set $s = S2.3$.

Step 1 In S2.3 there is one frond edge, namely, E−F.

Figure 11.14 DAGs obtained after deleting edge indicated in Figure 11.13.

Step 2.3 The weight of each node in S2.3 is 1.

Step 2.3 As both nodes in S2.3 have the same weight, either can be selected as the root. Let us select node `F` as the root. `Stubs` = `Stubs` ∪ {`F`}

Step 2.4 Set $G = S2.3$ and apply `TJJM`.

Step 1 There are two SCCs in the modified S2.3. Each of these is trivial and consists of, respectively, nodes `E` and `F`. Hence return from this call to `TJJM`.

There are no remaining non-trivial SCCs. Hence the algorithm terminates. We obtain `Stubs` = {`C`, `F`, `H`}. Figure 11.14 shows the DAG obtained after having removed the edges from the ORD in Figure 11.13 as described above. Applying topological sort first to the DAG rooted at `G` and then to the DAG rooted at `C` gives the following test sequence:

1. Test A using Stub(C)
2. Test E using Stub(F) and A
3. Test D using A and Stub(H)
4. Test F using E and D
5. Test B using Stub(C), D, and Stub(H)
6. Test G using F and B
7. Test H using B and Stub(C)
8. Test C using A, H, and E

It may be noted that the stubs created by applying the TJJM algorithm depend on the sequence in which the nodes are traversed when identifying

Stubs created by the TJJM algorithm depend on the sequence in which the nodes are traversed. Hence, the generated set of stubs is not unique.

the SCCs in Step 1. Exercise 11.9 asks for applying the algorithm with a different traversal order than used in Example 11.11.

11.6.3 The BLW method

The BLW method creates a test order with the goal of minimizing the number of stubs and specific stubs.

The BLW method aims to minimize the number of specific stubs. As explained in Section 11.5.3, minimizing specific stubs will likely reduce the total cost of integration testing by allowing the construction of simpler stubs, one for each client.

The BLW method is similar to the TJJM method in that it recursively computes SCCs and breaks cycles by removing edges. However, the strategy to remove cycles from an SCC differentiates it from the TJJM method. Recall that the TJJM method uses the following steps to break cycles in a non-trivial SCC:

The strategy used to remove cycles from a strongly connected component differentiates the BLW method from the TJJM method.

1. Compute the weight of each node in the SCC as the sum of the incoming and outgoing frond edges.
2. Identify the node with the highest weight as the new root, and a stub, of the SCC.
3. Delete all edges coming into the root node from within the SCC.

Instead, the BLW method uses the following steps to identify and remove edges from an SCC:

The BLW method does make use of the edge labels in an ORD.

1. Compute the weight of each association edge X−Y in the SCC as the product of the number of edges entering node `X` and the number of edges exiting node `Y`. Note that in contrast to the TJJM method, this step does not use the notion of fronds. Also, in this step the weight of an edge, not a node, is computed.
2. Delete the edge with the highest weight. In case there are multiple such edges, select any one. If the edge deleted is X−Y, then node `Y` is marked as a stub. The specific stub in this case is Stub(`X`, `Y`). Thus, instead of deleting all edges entering a selected node, only one edge is removed in this step. Note that only association edges are deleted. Again, this is in contrast to the TJJM method which could delete any type of edge.

All steps in the BLW algorithm are the same as shown in Procedure TJJM. The BLW algorithm makes use of ideas from the TD and TJJM methods. From the TD method it borrows the idea of computing edge weights. From the TJJM method it borrows the idea of recursively computing the SCCs

and deleting one association edge at a time. By doing so the BLW method gets the following advantages:

1. It creates all the stubs needed. This is unlike the TD method that may force the combined testing of some classes.
2. By aiming at reducing the number of specific stubs, it attempts to reduce the cost of an integration test order.

The following procedure is used in the BLW method for generating stubs. The test order and specific stubs can be determined from the resulting DAG and the stubs.

Inputs: (a) ORD G and (b) `Stubs`.
Outputs: (a) `Stubs` and (b) modified G with no cycles.

Procedure: `BLW`

Step 1 Find the set S of non-trivial strongly connected components in G. A non-trivial component is one with no cycles.

Step 2 Return if S is empty. Otherwise, for each component $s \in S$, do the following

2.1 For each association edge $X - Y \in s$ compute its weight as the product of the number of edges entering node `X` and the number of edges exiting node `Y` from within s.

2.2 Let $X - Y \in s$ be the edge with the highest weight. Remove X−Y from s. Set `Stubs` = `Stubs` $\cup \{Y\}$. In case there are multiple edges with the same highest weight, select one arbitrarily.

2.3 Set $G = s$ and apply procedure BLW. Note that this is a recursive step.

```
End of Procedure BLW
```

Procedure `BLW` is illustrated next with an example to find the stubs in the ORD in Figure 11.11.

Example 11.12 Let us apply procedure `BLW` to the ORD in Figure 11.11.

Step 1 The first application of Tarjan's algorithm as in Step A gives us two SCCs. The non-trivial SCC labeled `S1` is shown in Figure 11.15. Thus we have $S = \{S1\}$. There is only one trivial SCC in this case that consists of node `G`.

Step 2 Set $s = S1$.

Figure 11.15 SCCs, deleted edges, and stubs obtained by applying the BLW algorithm to the ORD in Figure 11.13. Only the non-trivial SCCs, i.e., those with one or more cycles, are shown.

Step 2.1 The weights of various edges in s are as follows:

H−B =	(3*3) 9	C−H =	(3*2) 6	A−C=	(3*3) 9
B−C =	(1*3) 3	E−A =	(2*1) 2	C−A =	(3*1) 3
F−D =	(1*2) 2	B−D =	(1*2) 2	E−F =	(2*2) 4
C−E =	(3*3) 6				

Step 2.2 The underlined edges H−B and A−C have the highest weights. H−B is arbitrarily selected and removed from s. Hence, node `B` is marked as a stub. `Stubs` = `Stubs` ∪ {`B`}.

Step 2.3 Set $G = s$ and execute procedure `BLW`.

Step 1 We get two SCCs only one of which is non-trivial and is labeled `S2` in Figure 11.15. The trivial SCC consists of only node `B`.
Step 2 Set $s = S1$.

Step 2.1 The weights of various edges in s are as follows:

F−D =	(1*2) 2	E−F =	(2*2) 4	C−E =	(2*2) 4
C−H =	(2*1) 2	E−A =	(2*1) 2	C−A =	(2*1) 2
A−C =	(3*3) 9				

Step 2.2 Edge A−C has the highest weight and is removed from s. Hence, node `C` is marked as a stub. `Stubs` = `Stubs` ∪ {`C`}.
Step 2.3 Set $G = s$ and execute procedure `BLW`.

Step 1 Once again we get two SCCs only one of which is non-trivial and is shown as `S3` in Figure 11.15. The trivial SCC consists of the lone node `A`.
Step 2 Set $s = S3$.

Step 2.1 The weights of various edges in s are as follows:

E−F =	(2*2) 4	C−E =	(1*1) 1	C−E =	(2*2) 4
C−H =	(1*1) 1	F−D =	(1*1) 1		

Step 2.2 Edge E−F has the highest weight and is removed from s. Hence, node `F` is marked as a stub. `Stubs` = `Stubs` ∪ {`F`}.
Step 2.3 Set $G = s$ and execute procedure `BLW`.

Step 1 We get four SCCs only one of which is non-trivial and is labeled `S4` in Figure 11.15. The trivial SCCs are {`E`}, {`D`}, and {`F`}.
Step 2 Set $s = S4$.

Step 2.1 The weight of the lone edge C−H in s is (1*1)=1.
Step 2.2 Edge C−H obviously has the highest weight and is removed from s. Hence, node `H` is marked as a stub. `Stubs` = `Stubs` ∪ {`F`}.
Step 2.3 Set $G = s$ and execute procedure `BLW`.

The algorithm will now terminate after returning from various recursive calls as there are no remaining non-trivial SCCs. The DAG obtained after deleting the edges as above is shown in Figure 11.16. The DAG, together with the class relationships in

Figure 11.16 DAG obtained by applying procedure BLW to the ORD in Figure 11.11.

Figure 11.11, can now be traversed to obtain the following test order:

1. Test A using Stub(C)
2. Test E using Stub(F) and A
3. Test C using A, Stub(H), and E
4. Test H using Stub(B) and C
5. Test D using A and H
6. Test F using E and D
7. Test B using C, D, and H
8. Test G using F and B

A total of four specific stubs are needed. These are Stub(`C`, `A`), Stub(`F`, `A`), Stub(`H`, `E`), and Stub(`B`, `C`).

11.6.4 Comparison of TD, TJJM, and the BLW methods

Algorithms for generating a test order from an ORD can be compared based on the cost of the generated stubs and the time to generate a test order.

Two criteria are useful in comparing the utility and effectiveness of the methods for generating a test order. First criterion is the number of stubs and specific stubs needed. As mentioned earlier, the cost of integration testing is proportional to the number of specific stubs needed and hence this seems to be a reasonable criterion. The second criterion is the time required to generate the test order given an ORD.

Several ORDs are used to compare the different methods for test order generation using the criteria mentioned above. Some of these ORDs are

from the literature on integration testing while others were generated randomly as described next.

Generating a random ORD: Table 11.1 lists three sets of ORDs used for comparing TD, TJJM, and BLW. The first set of three ORDs, labeled TDG, TJJMG, and BLWG, are from the publications where they were first reported. The second set of four ORDs are derived randomly and so are the ORDs in the third set. ORDs in Set 2 are relatively small, each consisting of 15 classes. Those in Set 3 are relatively large, each consisting of 100 classes.

The comparison was done using randomly generated ORDs as large as containing 100 nodes.

The generation process ensures that a randomly generated ORD has no cycle involving the inheritance and aggregation edges. Also, the generated ORDs do not have multiple inheritance. Each randomly generated ORD has the following parameters:

- number of nodes (n),
- density (between 0 and 1), where 0 means the ORD has no edges and 1 means that it is completely connected, and
- the relative proportions of the inheritance, aggregation, and association edges.

The density of an ORD indicates the extent of connectedness among its nodes. ORDs with different densities were constructed.

As an example, an ORD with 15 nodes can have at most $15 * (14) = 210$ edges assuming that only one type of edge exists in a given direction between any two nodes. Thus, a density of 0.5 indicates that the ORD has 105 edges. An ORD with, for example, 15 nodes and 43 edges is named N15E43. Note that the density of an ORD corresponds to the coupling between classes; the higher the density, the higher the coupling. All ORDs in Set 2 and Set 3 were generated with the relative proportions of inheritance, aggregation, and association edges set to, respectively, 10%, 30%, and 60%. Next, the three methods are compared against each other using the criteria mentioned earlier.

Criterion 1: Stubs and specific stubs: The performance of the three test order generation algorithms was compared against each other by applying them to various ORDs listed in the column labeled **ORD** in Table 11.1. The following observations are from the data in Table 11.1:

In the experiments the TJJM method was found to generate a slightly fewer number of stubs than the BLW method.

Observation 1: Among the TJJM and the BLW methods, the former creates the smallest number of stubs. This happens because in Step 2.3 in Procedure TJJM, all edges coming into the selected node from within an SCC are deleted. This leads to a quicker reduction in the number of cycles in the ORD and hence fewer stubs. In contrast, procedure BLW removes one edge at a time thus leading to slower breaking of cycles and generating a larger number of stubs.

In the experiments the BLW method was found to generate a slightly fewer number of specific stubs than the TJJM method.

Table 11.1 Performance of procedures TD, TJJM, and BLW for generating an integration test order.

Set	ORD	Nodes (Density)	Method Stubs (Specific stubs)		
			TD	TJJM	BLW
1	TDG	15 (0.12)	4 (4)	4 (4)	4 (4)
	TJJMG	12 (0.15)	4 (4)	4 (6)	4 (4)
	BLWG	8 (0.3)	5 (5)	4 (6)	4 (4)
2	N15E42	15 (0.20)	6 (7)	4 (7)	4 (6)
	N15E78	15 (0.36)	10 (11)	9 (30)	10 (19)
	N15E99	15 (0.72)	14 (17)	11 (58)	15 (49)
	N15E134	15 (0.90)	15 (16)	11 (63)	12 (48)
3	N100E1452	100 (0.16)	65 (106)	81 (711)	91 (589)
	N100E3044	100 (0.30)	51 (118)	86 (1431)	95 (1258)
	N100E4581	100 (0.45)	68 (141)	92 (2208)	99 (1986)
	N100E5925	100 (0.60)	97 (152)	95 (2948)	98 (2768)

In the experiments the density of the ORD had a significant impact on the number of stubs and specific stubs generated.

Observation 2: Among the TJJM and the BLW methods, the latter creates the least number of specific stubs. This indeed is the goal of the BLW method and achieves it by selecting the edges to be deleted in each recursive step using a method that is different from that used by TJJM.

Observation 3: As might be expected, the number of stubs and specific stubs created increases with the density of an ORD with N100E3044 being the only exception in the TD method.

In the experiments the number of stubs generated by the TD method was found to generally smaller than that generated by the other two methods as the size of the ORD increased. This happens because the TD method clusters together several classes thus reducing the number of stubs and specific stubs.

Observation 4: The density of an ORD has a significant impact on the number of stubs and specific stubs generated by the TJJM and BLW methods. For example, in Set 2 the number of stubs increases approximately three times as the density increases from 0.2 to 0.9. Correspondingly there is almost a sixfold increase in the number of specific stubs needed. The increases are not as dramatic in the TD method.

Observation 5: The number of stubs as well as the specific stubs created by the TD method gradually becomes much smaller when compared against those created by the other two methods. This result might be surprising though an in-depth understanding of procedure TD indicates that the number of stubs generated reduces with the size of the ORD. The following example provides an explanation.

Example 11.13 Consider a fully connected ORD with six nodes labeled `A` through `F` and only association edges. Procedures TJJM

and BLW each generate 5 stubs and 15 specific stubs. Procedure TD creates only two stubs and nine specific stubs. To understand why, examine the following test order generated by procedure TD:

Test classes `C`, `D`, `E`, and `F` as a cluster using stubs for `A` and `B`.
Test class B using stub for `A`.
Test class A.

Nodes `C`, `D`, `E`, and `F` are clustered because they are assigned the same major and minor levels, namely, 1. Each of these four nodes when tested against the two stubs leads to eight specific stubs. In addition, testing `B` with stub `A` leads to the ninth specific stub. Of course, a critical question is "Why does Procedure `TD` not break cycles in this cluster?" The answer to this question lies in Step 1 of Procedure `breakCycles`. Exercises 11.18 and 11.19 should help answer this question.

Criterion 2: Time to generate a test order: It is worth looking at the time needed to generate a test order and how it is impacted by the size and the density of the ORD under consideration. It is obvious that the time to generate a test order will increase with the number of nodes and density. However, in an experiment with 100 nodes and a density of 0.6, procedure BLW generated a test order in 1.29 minutes. The corresponding times for procedures TD and TJJM are 8.79 and 0.73 seconds. All times were measured on an iMac running on Intel Core 2 duo at 2.93 Ghz with 4 GB of main memory.

The time to generate a test order appears to be of no practical significance.

Given the times mentioned above, and the fact that not many software designs will be as large and as tightly coupled as in the experiment, the performance of the three algorithms should not be of any concern to a tester.

11.6.5 Which test order algorithm to select?

From the above discussion it is clear that the BLW method would be the best choice when specific stubs are important. This could happen in cases where two or more classes need to use a stub in different ways thus requiring multiple stubs for the same class. When this is not an issue, the TJJM method seems to be a better choice as it leads to a fewer stubs as compared to the BLW method.

The TJJM method is preferred when a minimal number os stubs is desired. The BLW method could be preferred otherwise.

In either case, the number of stubs generated by the BLW method is close to that generated by the TJJM method. For example, for a 15 node ORD, the number of stubs generated by applying the TJJM method ranges

from 4 to 11 as the density increases from 0.1 to 0.9. The corresponding numbers for the BLW method are 4 to 12. Thus, at least for relatively small ORDs, the BLW method seems to be a better choice than the TJJM method regardless of whether or not specific stubs are important.

The TD method appears to be useful when clusters of classes need to be tested together rather than tested in a purely incremental manner.

The TD method seems to be useful when one needs clusters of classes to be tested together. For low density ORDs, as in Set 1 in Table 11.1, the TD method is unlikely to create node clusters. Also, for such ORDs the number of stubs generated is close to those generated by the other two methods. However, as the ORD size and density increases, the TD method loses out to the other two methods.

11.7 Test Generation

11.7.1 Data variety

The primary objective of tests generated for integration testing is to test the connections across various components that have been integrated.

As mentioned earlier, in integration testing the focus is on the connections between components that have been integrated. Thus, tests generated during integration testing ought to ensure that the connections are tested adequately. Certainly, any of the test generation methods introduced in earlier chapters can be used to generate tests for integration testing. However, the objective of tests generated must be a thorough test of the connections, and not necessarily the remainder of the code in each unit. The next example illustrates the focus of integration tests.

Example 11.14 Figure 11.17 shows a partial set of classes in a university academic records management application. Consider a class named `Graduate` that contains methods to evaluate whether or not a student is eligible for graduation. Such a class might be a part of a larger system used at a university for academic records management. The `Graduate` class accesses student data through another class named `Srecord`.

The `Srecord` class contains methods to obtain student data from a database maintained by the university. An object of type `Graduate` asks another object of type `Srecord` to *get( )* the data for a student. Let us assume this is done inside a loop that accesses all students in the database. Based on the information in the student record, the `Graduate` object asks another object named `requirements` for the requirements to be met to graduate with a certain major. An evaluation is carried out upon receiving the necessary information. The outcome of the evaluation, e.g., whether or not the student is eligible to graduate, is written into the database via a call to the *put( )* method in the `Srecord` object.

Figure 11.17 A subset of classes in a student record management system. The arrows indicate the direction of data flow. Simply reverse the arrows to obtain the ORD relationships described earlier. *get( )* and *put( )* are methods to read and write data.

Assume that each of the three classes have been tested individually with any needed stubs. When testing the subsystem consisting of the three classes in Figure 11.17 the focus must be on the connections between them. These connections are via the *get( )* and *put( )* methods. It is obvious that during unit testing these methods must have been called assuming that a minimal level of code coverage, e.g., statement coverage, was obtained when running the unit tests. So what needs to be tested when testing the integrated set of classes?

The focus now is on the different types of data that each invocation of *get( )* might return as well as the different types of data that are passed to *put( )* for database update. During the unit testing of `Graduate`, it is possible that the stub used for `Srecord` might simply return one student record of a given type. In integration testing the tests must ensure that `Graduate` is tested against all possible types of student data. In addition, there might be exceptions generated when invoking *get( )* on either `Srecord` and `Requirements`. Tests must ensure that such exceptions are generated and handled appropriately by `Graduate`. Testing against exceptions might require some form of simulation of the database or simply the disconnection of the database from `Srecord`.

Coverage criteria based on the number of connections covered, and their type, can be a useful test adequacy criterion during integration testing.

Focusing on the variety of data processed in each of the classes integrated will likely lead to better coverage of the code. Note that a number of different types of student profiles can be generated for tests. Combinatorial design techniques for test data generation is one potential technique to generate data during this integration test. While such tests could be generated during unit testing, doing so would be difficult, or even impossible, when using stubs that are simplified versions of the real classes they represent. Thus, faults that were not revealed during unit testing might be revealed during an integration test.

Combinatorial design techniques can be used to generate integration tests.

11.7.2 Data constraints

Errors that lead to incorrect calls to methods across components might not be revealed during unit testing.

The example above illustrates how new tests need to be created during integration testing using variety of data. In some cases errors related to the incorrect calls to methods across components being integrated might not be revealed during unit tests. Simply making calls to such methods during integration testing might reveal such errors. The next example illustrates one such case.

Example 11.15 Consider two objects named `X` and `Y`. Suppose that `Y` contains a method `m1()` that takes two integer parameters a and b as inputs. The two parameters must satisfy a relationship, such as $a < b$. However, when `X` is tested with a stub for `Y`, there is no check in the stub among the relationship between parameters a and b. Thus, any error in `Y` that affects the relationship between the two parameters might remain undetected during unit testing. However, when `X` and `Y` are integrated such an error will likely be detected if a variety of data values are generated for the parameters. Such tests could be generated by suitably using, for example, the BRO method (see Chapter 4). Doing so will ensure that the relation $a < b$ is false at least in one test. This test may reveal an error in `X` that relates to either the generation of the values of a and b, or its reaction to the response given by `Y` when an incorrect set of parameters is passed to method `m1()`.

Errors not revealed during unit tests have good chances of being revealed during integration testing when the focus of the tests is on the connections across the various components.

It is perhaps now clear that errors that are not revealed during unit tests might be revealed during integration tests if tests are generated with a focus on the connections between the components being integrated. However, the generation of such tests is not always easy. The unit tests created for a component are not likely to be adequate for integration testing for at least two reasons. First, the component being integrated might be hidden inside the subsystem under test and hence its unit tests are not valid any longer. For example, tests created during unit testing of `Srecord` in Figure 11.17 cannot be used when testing together with `Graduate`. This is because `Srecord` is now driven by `Graduate` and not a driver that might have been used when it was unit tested. Second, the tests generated during unit testing focus primarily on the unit under test. Hence, such tests might be incomplete from the point of various test adequacy criteria discussed in the next section and in more detail in chapters under Part 3.

11.8 Test Assessment

Adequacy of a test set used for integration tests can be assessed in a variety of ways. One may use black-box as well as white-box adequacy criteria for such assessment. Black-box criteria are based upon requirements and any formal specifications. White-box criteria are based on the application code. In any case, the criteria need to be applied to the integrated subsystem and not to the units. White-box criteria are based on control flow, data flow, and mutation. These are discussed in detail in chapters under Part 3. Note that these criteria are generic in the sense that they can be applied during both unit and integration testing. However, when applied during integration testing, the criteria are defined on program elements that correspond to the connections between components and not to the entire unit. The following examples briefly illustrate a few such criteria.

A variety of test adequacy criteria discussed in earlier chapters can be used as well during integration testing. However, it is important to refocus such criteria on the program elements that relate to connections across components and ignore those that do not.

Example 11.16 Suppose that an application to generate random graphs is to be tested. Components of one such application are shown in Figure 11.18. There are several ways these components can be integrated and tested.

Now consider the following requirement: *The application must get from the user the relative proportions of the inheritance, aggregation, and association edges.* It is best to generate several tests to test for this requirement. Some of these tests would be extreme, e.g., all edges of one type. Others might be more moderate, e.g., with relative proportions of edges being 0.1, 0.3, and 0.7. Yet, another test could be for robustness where the sum of the proportions entered by the user is not equal to 1. Note that these tests will check whether or not the component that inputs the data and those that actually use it to generate the graph are behaving as expected. A

Figure 11.18 Components and their relationships in an application to generate a graph randomly. The generated graph is assumed to be an ORD with only association edges.

simple requirements-based adequacy criterion is satisfied when the application was found to behave as expected on all tests.

Example 11.17 Suppose two components `C1` and `C2` are being tested. As shown in Figure 11.19, `C1` has a method `m1` that invoked method `m2()` in `C2`. Let us now examine the various data flows in this simple example. The input parameter `x` is tested in `m1()` and the value of `p` computed and returned. Now suppose that when testing `C1` with a stub of `C2`, t `m2()` always returns -1. Two tests of `C1`, one with $x < 0$ and the other with $x \geq 0$, would be adequate to cover all paths in `m2()`. However, doing so would not test thoroughly the logic for $x \geq 0$.

When `C1` and `C2` are integrated, it is important that the path labeled P1 in Figure 11.19 be tested in conjunction with at least path P2 in `m2()`. Doing so causes the definition–use pair consisting of the definition `x` in `m1()` and its use in `m2()` to be covered. Such definition–use pairs, also known as *d–u* pairs, across components under test constitute a class of elements on which an adequacy criteria can be based.

Figure 11.19 Integration of components `C1` and `C2`. Function `f` takes a double value as input and returns a value of the expected type.

In this example, there are two such d–u pairs across the components. One pair consists of the definition of `x` in `m1()` upon entry and its use in the condition in `m2()`. The other d–u pair consists of the same definition of `x` paired with its use in `m2()` where function f is computed. Any test for which $x \geq 0$ will cover both these pairs when testing `C1` and `C2`. This flow of data that covers both the d–u pairs is shown by the doted lines labeled 1 that is followed by the dotted line 2.2. The dotted lines 1 followed by 2.1 do not cover any d–u pair. (Also see Exercise 11.21.)

Example 11.18 In Figure 11.19 each method has two paths corresponding to the respective conditions in the code. Thus, when combined, we get four possible paths to be tested. However, assuming that unit testing has covered all paths within each method, one needs to test only the path P1 combined with paths P2 and P3. Thus, any test that contains two test cases, one in which $x = 0$ and the other in which $x \neq 0$, will cover these two combinations of paths across the two components.

It is possible to reduce the number of paths to be tested during integration testing if the paths covered during unit testing of the connected components are removed from consideration.

Identifying data flows and control paths across components leads to the construction of white-box coverage criteria. One or more of such criteria could be used during integration testing to assess the goodness of tests (see Exercise 11.22). Such data and control flow based criteria, as well as another technique known as *interface mutation*, are described in detail in Part 3.

Control flow, data flow and mutation based test assessment techniques are applicable during integration testing.

11.9 Tools

`Tool`: inTegToolset
`Link`: http://www.cs.purdue.edu/homes/apm/foundationsBook/
`Description`

This is a collection of four tools developed by the author. Each tool uses a specific algorithm to generate an integration sequence from a given class dependence graph. The tools, named after the inventors of the algorithm, are `BLWAlgorithm`, `TJJMAlgorithm`, and `TDAlgorithm`. A fourth tool is named `CreateRandomORD` that is used to generate random ORDs for testing the effectiveness of a test order generation algorithm. The code for all four tools is freely downloadable.

`Tool:` EasyMock
`Link:` http://www.easymock.org

`Tool:` jMock
`Link:` http://www.jmock.org
`Description`

The above two tools are frameworks for creating mock objects. jMock offers a language for specifying the behavior of a mock object. This language is named Domain Specific Language (DSL). EasyMock uses a capture–replay approach for specifying the behavior of a mock object. Thus, the mock object is first trained by making the expected method calls. Then, in the replay mode during test of a system, the mock object behaves as desired.

`Tool:` JSystem
`Link:` http://www.jsystemtest.org/
`Description`

This tool is a framework for automated system testing. It has features that allow a tester to write and automate tests, create scenarios, run tests on a remote server, and obtain test statistics. It uses the JRunner tool to run tests.

SUMMARY

Integration testing is an intermediate phase in the software development process. During this phase, testers decide on a suitable order for integrating and testing various components of a system. The focus in this phase is to uncover any errors that might arise when two or more components work together. It is assumed that such errors are not discovered during unit testing.

This chapter begins by explaining integration errors. It then enumerates various types of inter-component dependences and how these are represented in graphical form. The notion of an Object Relation Diagram (ORD) is introduced. The three different types of integration strategies, namely, top-down, bottom-up, and mixed, are then described and their pros and cons considered. This background material is then followed by a rather long section that describes three algorithms for generating a test order during integration testing. A comparison of these three algorithms using randomly generated ORDs is also provided. In the end, the processes of test generation and test adequacy assessment during integration testing are discussed. Note that the foundations of test adequacy assessment are laid out in the chapters under Part III.

Exercises

11.1 List three test sequences for the components in Figure 5.20 when using (a) top-down testing strategy and (b) bottom-up testing strategy. In each case list the stubs or drivers required.

Figure 11.20 A hypothetical component hierarchy for Exercise 11.1. Components C6 and C7 extract data from a database.

11.2 Compute the firewall for each node in Figure 11.5(b). Do you notice any differences in the firewalls computed in Example 11.4? Explain the difference or a lack thereof.

11.3 List at least two reasons why an ORD created using UML design diagrams might differ from the one created using the corresponding application code?

11.4 Suggest how you might proceed to create an ORD when a complete application design is available but the code is available only for some of the classes mentioned in the design.

11.5 (a) List a test order for the classes in Program P11.1 such that no stubs are needed. (b) List the class firewall for each class in Figure 11.6(a)? Does the firewall for any class change if it were instead derived from Figure 11.6(b)?

11.6 Let O1 and O2 be two ORDs, O1 being more precise than O2. What can you say about the number of edges in O1 as compared to the number of edges in O2?

11.7 Create two class test sequences from the ORD in Figure 11.21. One sequence must use stubs for classes `B`, `C`, and `F` as stubs while the other must use stubs for classes `E`, `D`, `C`, and `H`. Compare the number of specific stubs required of each test sequence.

Figure 11.21 A sample ORD with cycles.

11.8 Answer the following with reference to the ORD in Figure 11.22. (a) Find a minimal set of stubs $S1$ and the corresponding incremental test sequence. How many specific stubs are required if $S1$ is to be used? (b) Find a set of stubs $S2$ that minimizes the number of specific stubs. What is the incremental test sequence if $S2$ is to be used during testing?

Figure 11.22 A sample ORD with five classes containing cycles between classes `A`, `D` and `D`, `E`. Dashed arrows indicate potential stubs.

11.9 How does, if any, the choice of the first node input to procedure `assignMajorLevel` on page 639 might affect the integration sequence output by the TD method?

11.10 In the TD method no two nodes at the same major level can be connected by an inheritance or an aggregation edge. Why?

11.11 Complete Example 11.8.

11.12 Complete Example 11.9.

11.13 Complete Example 11.10.

11.14 The TJJM algorithm ignores the types of class dependence such as aggregation, association, and inheritance. Justify this assumption.

11.15 In Example 11.11 why did we apply topological sort first to the DAG rooted at node `G` in Figure 11.14?

11.16 Redo Example 11.11 but assume that node `B` is traversed prior to traversing node `F`. The nodes in SCC S2 are thus traversed in the following sequence: `B`, `H`, `C`, `A`, `A`, `E`, `F`, and `D`. What differences do you observe in the outcomes of this exercise and those of Example 11.11?

11.17 At Step 2.2 in Procedure `BLW`, more than one edge might have the same highest weight. Show that the choice of which of these edges selected for deletion does not affect the number of specific stubs generated.

11.18 In Example 11.13 we observed that Procedure `breakCycles` failed to break the cycles in a subgraph. This led to a test order in which four nodes are to be tested together. Examine Step 1 in Procedure `breakCycles` and explain why the cycle is not broken.

11.19 Show that for a fully connected ORD having only association edges and with four or more nodes, the number of stubs generated by Procedure `TD` is 2.

11.20 Suggest a method for breaking cycles in clusters of classes formed when applying procedure `TD`. Apply your method to a fully connected ORD with six nodes and only association edges. Compare the number of stubs generated with that obtained by applying the TJJM and BLW methods.

11.21 (a) Apply the TD, TJJM, and BLW methods to generate test orders for the application shown in Figure 11.18. (b) Generate at least one test that is useful in each step of the test order. (c) For each step in the test order, indicate at least one white-box test adequacy criterion that you might use.

11.22 Suppose that components `C1` and `C2` contain, respectively, p_1 and p_2 paths. Assume that each of these paths is covered during unit testing. (a) Under what conditions it is not necessary to test all possible combinations of the paths when `C1` and `C2` are tested together? (b) What are the maximum and the minimum number of paths that might need to be tested? Provide situations where the maximum and minimum values are possible.

This page is intentionally left blank.

Index

⊗
  onto set product, 183
  partial string concatenation operator, 251
#define, 364
@Before, 609
@BeforeClass, 609
@Test, 609, 616
1-equivalence partition, 235–236
1-switch cover, 282, 285–286
  example, 286
2-equivalence partition, 237
3-way interaction fault, 306
  incorrect arithmetic function, 307
5ESS, 38
80x74, 86

A
ABP: Alternating Bit Protocol, 224
ABS operator, 466, 518
abstract data type, 90
abstract syntax tree, 74, 198–200
acceptance testing, 56–57
access control, 469
access modifier in Java, 469
activation request, 581
activity diagram, 41
ACTS, 342
adequacy
  requirements based, 665–666
  with respect to decision coverage, 373, 377, 398
adequacy assessment, 46, 53, 59, 93
  mutation based, 452–462
  scalability, 446
adequacy criteria
  based on control flow, 426
  based on data flow, 426
  black box, 351, 665
  control flow based, 364–402
  more powerful, 430
  non-code based, 430
  quantitative, 541
  weaker, 382
  white box, 351, 665
adequacy measurement, 351
  example, 351–353
AETG: Automatic Efficient Testcase Generator, 7, 342
aggregation, 625–626
aggregation edge, 659
agile
  development, 533
  testing, 55, 57, 64
AGTC: Applied Genomics Technology Center, 320
aircraft, 216
aircraft landing, 51
alarm, 178
algebraic languages, 41
all-uses, 419, 429, 535, 540
  coverage, 319, 535
alphabet, 98
  binary, 98
  input, 218, 220
  input of an FSM, 226
  output, 218
  output of an FSM, 226
AND-node, 165, 166, 174–176
  BRO constraint set, 188
ANSI C standard, 496
ANSI/IEEE, 10, 68
ANSI/IEEE Std, 10, 68, 729–1983
Apache, 431
Apple
  G4, 302
  G5, 302

applet, 315
application
  behavior, 44
  binary, 443
  carefully designed, 127
  commercial, 106
  deterministic, 330
  networked, 142
  non-deterministic behavior, 330
  payroll processing, 117
  requirements, 116
architecture visualization, 100
array, balanced, 328
artifact under test, 53
ASCII, 17
assembly language, 14, 364
assignment, 406
  missing, 539
association, 625
  field access, 626
  method call, 626
association edge, 626
  weight, 654
AT&T, 38
ATAC, 594, 595, 596
Athelon, 302
ATM, 212
atomic propositions, 45
author, 43
automaton, 218
automobile, 216
automobile engine controller, 552

B
backward trace, 169
balanced array, 328
balanced design, 313
banking software, 552
basic block, 74–76, 205, 365, 406, 558, 560–561, 587, 590
  example, 74–75
  exponentiation program, 74
  global and local use, 405–406
  number, 77
  unreachable, 365
  variables defined in, 406
basis path, 83
basis set, 83
batch processing, 53–54
batch processing applications, 53–54
behavior observable, 6–7
behavior verification, 609
beta testing, 49
Big-bang integration, 631
big-bang process, 30
billing transactions, 52
binary numbers, 98
binary strings, 313
BIST: Built-in Self Test, 35
black-box category, 54
black box criteria, 665
black-box testing, 47, 58
block coverage, 365, 426
  example, 366
block, *see* basic block
BLW: Briand-Labiche-Wang method, 637
BLW procedure, 655
boiler
  gas-fired, 69
  heating system, 125
  steam, 69
  temperature, 125
boiler command
  cancel, 125
  set temperature, 125
  shut down, 125
boiler control, 125
  control option, 125
  equivalence classing, 126
  error detection effectiveness, 149
  infinite values, 127
  shut command, 131
  test selection example, 129
  weaknesses in test, 148
Boolean expression, 160, 166
  conversion between CNF and DNF, 73
  examples, 72
  non-singular, 72
  singular, 72
Boolean operator, 72, 176, 367
  binary, 72
  incorrect, 176
  unary, 72
Boolean operator fault, 176–177, 202
  example, 177
  multiple, 210
Boolean and Relational Expression, 181
Boolean variable, 72, 73, 367, 390
Boot Camp, 302
bop: Boolean operator, 72
BOR adequate test set, 181, 199–200
  example, 181
BOR: Boolean OperatoR, 181
BOR constraint, 184

BOR constraint set
  example, 185
  generation, 184–185
border, 87, 154
  closed, 154
  open, 154
  segment, 87, 153
  shift, 153
  straight line, 87
BOR testing criterion, 181, 187
boundary
  identification, 133, 146
  tests near, 137
boundary interior, 159, 282–283, 441
boundary interior cover, 283
boundary-interior testing, 441
boundary-interior test paths, 159, 441
boundary point, 135
boundary shift, 154, 155
boundary value analysis, 70, 133–139, 146, 149, 153, 296, 618
  example, 134, 137
  procedure, 133
  test suite, 136
braces(f, g), 69
branch coverage, 433
branch decision coverage, 369
branches uncovered, 41
branch points, 433
Brazil, 478
BR: Boolean and Relational, 179, 181
BR constraint, 180
BRE adequate test set, 181, 199, 200
  size of, 200
BRE constraint set, 189–191
  derivation, 210
  example, 191
  generation, 190–191
  for relational expression, 189–190
BRO, 664
BRO adequate test set, 181, 210
  size of, 200
BRO: Boolean and Relational Operator, 181
BRO constraint set
  example, 210
  example for non-singular expressions, 195–196
  generation, 187–188
  non-singular expressions, 191–197
BRO testing criterion, 181
Browser
  Chrome, 595
  combinations, 302
  Firefox, 595
  Opera, 595
buffer
  dynamic allocation, 52
  overrun, 100
  size, 52
bug, 6, 67
Bugzilla, 39, 67
bva: boundary value analysis, 133–139

C
C#, 66, 97, 287–288, 434, 534, 595, 602, 605, 616, 623
C++, 31, 53, 90, 100, 120, 364, 434, 469, 620, 623, 630
  coverage measurement tools, 431
C, 364–367, 404, 532, 534, 542, 602
  coverage measurement tools, 431
call graph, 97–98
  cycle, 102
  dynamic, 98, 100
  exact, 98
  for integration testing, 630
  static, 98
  undecidable, 98
call-by-reference, 404, 406
call-by-value, 404
cardiac pacemaker, 54
CASE: Computer-Aided Software Engineering, 531
cash register, 150
category, 141
category partition, 46, 140–146
  example, 141
  method, 140–146
cauliflower, 112, 113
cause, 160
cause-effect graph, 160–176
  construction, 166
  creation, 163–167
  example, 164
  first step, 165
  heuristics, 173
  heuristics example, 172
  incremental strategy, 165
  notation, 161–163
  procedure, 160
  trace back, 172
cause-effect relations, 197
Celsius scale, 58
CFG, *see* control flow graph
  annotated with data flow information, 91
chain, *see* def-use chain

characterization set, 234–242
  example, 240
Chidamber, Shyam R., 32
child function, 561
choices, 140
Chow, T.S., 283
Chrome, 595
class
  client, 635
  coupling, 32
  firewall, 627
  number of children, 32
  parent, 521
  response set, 32
  server, 635
  weighted methods, 32
ClearCase, 67
client-server testing, 53
CMS: COBOL Mutation System, 534
CNF: Conjunctive Normal Form, 73
COBOL, 66, 534
COCOMO II, 66
code
  access to, 431
  element hierarchy, 432
  instrumentation, 431
  stable, 533
CodeCover, 100
code coverage, 64
  measurement, 431
code coverage criteria, 58
code coverage measurement, 431, 616
  impractical, 431
  practical aspects, 431
code fragment, borrowed from Kernighan and Ritchie, 482
code inspection, 35, 43
code-inspection, 43, 91
code inspector, 91
code measurement tool, 432
CodeSonar, 100
code testing, 53
coding phase, 48, 56
coding step, 214–215
coincidental correctness, 149, 159
color laserjet, 147
combinatorial design, 295–343, 663
  balanced, 330
  procedure, 303
  strength, 332
combinatorial explosion, 160, 172
combinatorial object, 303
command file, 125
comment lines, 143
communication protocol, 224
  control portion, 224–225
  data portion, 224–225
  as finite state machine, 224
compatibility testing, 50
Competant Programmer Hypothesis, 537
Competent Programmer Assumption, 472
compiler, 30
compiler testing, 53
complete factorial design, 345
completeness assumption not satisfied, 260
completeness of FSM, 259
complexity
  module, 43
  see also cyclomatic complexity
complexity metric
  in static testing, 43
component, 602, 623
component testing, 53
composition, 625
compound condition, 367, 368, 371, 388
compound predicate, 181, 420
compound type, 120
computation error, 90
condition, 367–368, 586
  within assignment, 369
  atomic, 367
  combinations, 375
  compound, 367, 368
  coupled, 368
  as decision, 367
  elementary, 367
  interpretation, 87
  nonlinear path, 89
  simple, 367
condition coverage, 371–374, 375–378
condition/decision coverage, 373–375, 382–383
  example, 374, 375
confidence, 38, 60
  false sense of, 61
configuration
  order, 163
  sample, 165
configuration space, 297–303
configuration testing, 46–47
conformance
  example, 261
  strong, 261
  weak, 261
conformance testing, 224–228
conjunctive normal form, 73
connected assumption, 259
connectionless transmission, 224

constant mutation, 467–468
constrained mutation, 531
constraint, 179
  example, 179
  false, 180
  partition, 180
  specification, 143
  true, 180
constraint propagation, 200
container program, 542
context coverage, 423–425
contract software, 49
control computer, 178
control dependence, 96–97, 623
  example, 97
control dependency, 577
control flow graph, 74–92, 406, 435, 560
  child, 560
  construction, 74
  edge, 76
  of function, 560
  main, 560
  node, 76
  pictorial representation, 77
control node
  direct, 577
  indirect, 577
control path, 667
control software, 124, 178
CORBA, 48
core behavior, 259–261
  example, 260–261
  of FSM, 259
correctness, 18–19
  coincidental, 149, 159
correctness proof, 18
cost criterion, 590
cost driver, 67
coupling effect, 471–474, 537
coupon, 140, 150
coverage, 351
  block, 390
  boundary interior, 159
  extent, 351
  lack of, 430
  LCSAJ, 381
  as percentage, 367
  requirements, 430
  residual, 590, 591
coverage based minimization, 586
coverage criteria, constraining of, 432
coverage data, enormous amount, 432
coverage domain, 351, 352, 356
  feasibility demonstration, 358
  identification of infeasible elements, 360
  infeasible, 358
coverage measurement
  cause for slowdown, 432
  incremental, 432
  insufficient disk space, 432
  for object oriented programs, 433
  reducing the number of modules, 432
  selective, 432
coverage measurement tools, 431–432
covering array, 303, 330
  different from orthogonal array, 330
  example, 331
  mixed level, 303
CPH: Competent Programmer Hypothesis, 472
CPU, 164
crash, 52
CSP, 534
currency converter, 618
c-use, 405
  adequacy does not imply p-use adequacy, 428
  coverage, 412, 416, 427
  coverage example, 417
  feasible, 438
  infeasible, 421
  minimal set, 412
  minimization, 412
  number of, 412, 415
c-use: computational use, 405
cycles break, 643
cyclomatic complexity, 30, 32, 66, 69, 83–85, 204
  exercise, 69

D
DAG, 650, 653, 655, 657, 658
danger signal, 447
Darlington Nuclear Generating Station, 447
DART, 7
data, 95
  variety, 663
database disconnect, 663
database systems, 431
data context
  table, 413
  coverage, 423
  uncovered, 424
data dependence, 95, 622
  example, 96

Data Dependence Graph, 95
  errors, 132
data entry mode, 27
data environment, 412
data flow, 667
  for arrays, 438
  augment tests, 273
data flow coverage, 426, 558, 595
data flow error, 92
data flow graph, 406–408, 409–412, 415, 423
  example, 407
  paths, 408
data flow testing, 46, 474
data race, 100
DBMonster, 7
DC: Decision Coverage, 382
DDG: Dynamic Dependence Graph, 573
DDG, *see* Data Dependence Graph
deadlock, 100
debugging, 18, 21, 49, 93
decision table, 167–171
  from cause effect graph, 165
  empty, 169
  entry, 167
  generation example, 169
  generation procedure, 168–169
  as a matrix, 167
decision, 367
  covered, 369
  implied in switch, 369
  outcomes, 367
  undefined outcome, 367
decision coverage, 369–371
  domain, 371
  example, 370
  fault detection effectiveness, 471
def-clear path, 408–409, 438
  existence, 410
defect
  classification, 38
  density, 32
  discovery, 38
  management, 38
  open, 39
  prediction, 38, 39
  prevention, 29, 38
  recording and reporting, 38
  reported after release, 29
  resolution, 39
  saturation, 33
  severity, 32
  statistical techniques for prediction, 39
  tracking, 67
  types, 39
defects, 6
  cumulative count, 33
definition
  alternating, 410
  global, 406
  output variable, 412
definition clear path, *see* def-clear path
definition clear sub-path, 408
definition killed, 91, 406
definition-use pair, 586
def set, 407
def-use
  alternating, 410
  error detection example, 426
  reduction, 432
  sequence, 410
def-use chain, 410–411, 421
  example, 411
def-use graph, 406, 411, 422
  see also data flow graph
def-use pair, 409–410, 411
  example, 410
Dell Dimension Series, 302
DeMillo, Richard A., 478
dependence
  abstract, 630
  control, 623, 630
  data, 623, 630
  dynamic, 626–627
dependence graph, 74, 102
dependency modeling, 160
depends relation, 627, 628
deposit, 553
depth first search, 639
DES: Distinguishing Execution
    Sequence, 484
design
  balanced, 313
  complexity, 31
  high level, 106
  unbalanced, 331
design phase, 214
design testing, 53
design verification test, 215
developer, 7
development cycle, 38
development process, 602
dial-up connection, 296
Dieseltest, 7
DIGDEC machine, 217, 220–224
digital cross connect, 24
directed graph, 219
disjunctive normal form, 73
distinguished lady, 463

distinguished professor, 463
distinguishing sequence, 241
  brute force procedure, 289
distinguishing sequence method, 286
distinguishing signature, 273–275
  example, 274–275
DNA sequencing, 320
DNF: Disjunctive Normal Form, 73
DO 178B, 100, 533
dog food dispenser, 208
dog sitter, 208
domain
  coverage, 467
  error, 90
  testing, 46, 90, 91
dominator, 93–95
  example, 94
  immediate, 94
  relation, 94
  strict, 94
  tree, 94
Dom, *see* dominator
driver, 631, 632, 634
  construction, 633
  number of, 632
DS: Distinguishing Sequence, 286
DSL: Domain Specific Language, 668
d-u pair, 666–667
dynamic slice, 572–574, 581, 582, 585
  computation, 572–574
  precise, 572
  precision, 572
  statement addition, 580
  statement deletion, 581

E
EasyMock, 609–611, 615, 618, 667
  create mock object, 615
  expect, 615
  nice mock, 615
  strict mock, 615
Echelon, 7, 594–595
EclipseCFG, 100
E constraint, 162
edge, 407
  aggregation, 659
  association, 671
  frond, 650
  inheritance, 659
  outgoing, 410
  relative proportion, 659
effect, 160
  identification, 165
effort, 31
EFSM: Extended Finite State Machine, 288
Eggplant, 7
elementary data context, 413, 423
  covered, 424
  cumulative coverage, 425
  example, 413
  feasible, 440
elementary data context coverage, 424
elevator
  control, 34
  hoist motor, 34
  response, 34
  scheduling, 34
eLoadExpert, 7
embedded
  computer, 216
  system, 216
embedded environment, 594
embedded microprocessor, 208
embedded software, 552
embedded system
  coverage measurement, 432
  memory constraints, 433
emergency shutdown, 447
EMMA, 617
empirical studies, 526
employee record, 108
End node, 408
engine control, 216
Enter button, 27
entry point, 74–75, 365
environment, 216
environmental disaster, 448
environment object, 140
equality symbol (=), 176
equivalence
  boundary, 133
  class, 109, 116–121, 123, 124, 126
  combining, 124, 139
  cover, 130
  discard infeasible, 127
  don't care values, 128–130
  enumerations, 119
  example, 120
  explosion, 130
  infeasible, 124
  large number of, 130
  output, 137
  product, 130
  range, 117
  strings, 119
  testable, 128
  unmanageable size, 124

equivalence partitioning, 46, 53–54, 70, 103, 107, 109–133
effectiveness, 111
example, 124
GUI, 132
guidelines, 117
multidimensional, 121
procedure, 123–128
unidimensional, 122, 296
equivalence relation
reflexive, 147–148
symmetric, 147–148
transitive, 147–148
equivalent mutant, 459–461
error, 5–7
action, 221
computation, 90
domain, 90
integration, 620, 631
location, 635
masking, 202
missing path, 91
path selection, 91
removal, 18
error detection
extra state, 248
process, 249
transfer error, 249–250
error free program, 353
error message, 141, 148
as an effect, 160
error revealing test, 477–478
errors, 4
domain, 106
semantic, 5
syntax, 5
error seeding, 541
error studies, 471
ESG: Event Sequence Graph, 47, 50
Estelle, 224
event
external, 562
event-response pair, 206
executable entity, 590
execution
history, 92
exponentiation program, 92
space, 93
slice, 92
trace, 92
execution sequence
effect of, 484
example, 482
execution slice, 92, 560, 571, 572, 575, 583
execution trace, 560–570
collection, 562
example, 562
exhaustive testing, 298
exit, 552
point, 74–75, 365
expected behavior, 11
expected response, 246
exploratory testing, 58
Explorer, 287, 288, 302, 320, 345
exponentiation program, 74
flow graph, 80
extra Boolean variable, 178–179, 211
extra state error, 229, 248–249

F
f, 179
factor, 297, 298
binary, 328
for browser, 302
combination, 298
don't care, 307
in GUI, 299
identification, 302–303
levels, 298
for sort, 299
factor combination, 304
infeasible, 302
procedure for generation, 321–325
selecting a subset, 312
factorial, 261
factor level
unused, 300
used, 300
Fahrenheit scale, 58
failure, 5–6
count, 59
network, 143
false, 72
fault, 5–6
arithmetic expression, 177, 181
Boolean operator, 177, 179, 181
Boolean variable, 178
coverage, 233
dynamic binding, 522
extra Boolean variable, 178
extra state, 229
incorrect Boolean operator, 396
incorrect loop termination, 468
interaction, 298, 305
missing assignment, 539
missing Boolean variable, 179, 211
missing condition, 396

missing condition, detection using mutation, 474
missing initialization, 539
missing initialization, detection using static analysis, 540
missing state, 230, 290
missing statement, 473
multiple, 177
off-by-ϵ^*, 177
off-by-ϵ^+, 177
off-by-ϵ, 177
operation, 229
pairwise interaction, 306
polymorphism, 522
precedence, 495
propagation, 198
relational operator, 177, 181
sequencing, 230
simple, 305
single, 177
t-factor, 305
transfer, 229
t-way interaction, 305
vector, 307
fault classes, 177
fault detection, 287
comparison example, 283
using mutation, 474–477
fault detection effectiveness, 110, 427
of W-method, 282
of Wp-method, 282
fault-injection, run-time, 443
fault model, 34
for finite state machines, 228
fault propagation, example, 199
fault revealing test, 198
faults mixed kind, 396
fault vector, 307–308
faulty predicate, 182
features advertised, 46
FIND program, 540
finite state machine, 40, 216–224
1 distinguishable, 284–285
complete, 234
completely specified, 222, 242
connected, 242
core, 260
core behavior, 259–260
deterministic, 242
equivalence, 222
exciting, 245
initial state, 242
k-equivalence, 223
minimal, 223, 234, 242
strongly connected, 222
tabular representation, 221–222
V-equivalence, 222
finite state model, 44
Firefox, 67, 302, 320, 595
firewall, 627–628, 669
testing, 627
flight controller, 216
flow
of control, 364, 402
of data, 402
graph, 74, 82
flow of control, 74
FogBugz, 39
Foreign language testing, 50
formal method, 40
nuclear generating station, 447
formal model, 40
Fortran, 368, 443, 467, 469, 526
Fortran code, 446
frond edge, 650, 651
FSM, *see* finite state machine excitation
FSM testing, 46
functional requirements, 350
functional testing, 65–66, 429–431
functional unit, 131, 133, 135
function call, 76, 405, 479, 563–564, 567
function coverage, 432, 594
function trace, 584, 599

G
giveaway items, 164
glass-box testing, 429
global definition, 406
Goldsmiths College, 516
goodbye message, 361
goodness of a test set, 233, 350, 470
GOTO statement, 69, 502
gprof, 100
graph, 74
call, 97
graphical notation, 41
graphical user interface, 114, 131, 299
greatest common divisor, 86, 296
greedy algorithm, 587–588, 590
Grinder, 7
grocery item, 112, 113, 141
guard, 83
GUI, 124–127, 131–133, 176, 364
boiler control, 124
design, 133
dissection from application, 132
factors, 298
front end, 132

invalid inputs, 124
reduction in test cases, 133
test generation using factors, 304
testing, 124
GUI testing, 7, 58

H
Halstead, Maurice, 30
Halstead metrics, 30
Hamlet's program, 540
hand calculation, 26
hardware configuration
Dell Dimension Series, 302
hardware platform, 296–297
heart condition, 332
heart rate, 333
heart simulator, 552
Hierons, Rob, 150
home video, 27
HP, 66
human disaster, 448
human errors, 4
human tester, 47

I
IBM, 54
Rational ClearCase, 67
IBM/Rational, 91
ICD: Interface Control Document, 623
I constraint, 162, 196
iDatify, 206
community, 206
question, 206
identifier
external, 485
identity matrix, 26
IDL: Interface Definition Language, 48
IEEE, 33
if statement, 115
incorrect program, execution of, 401
incorrect scoping in C, 469
India, 551
infeasibility and reachability, 398
infeasible
basic block, 406
combinations, 160
condition, 389
constraint, 179
decision, 398–399
elementary data context, 440
use, 422
example, 422
infeasible data, 124
inheritance, 625
inheritance, multiple, 659
inheritence edge, 659
input
failure causing, 32
illegal, 110, 120
invalid, 17, 111, 133
isolation, 142
legal, 12, 108, 110–111, 117–119
palette of valid, 124
regions, 111
unintended, 51
valid, 17
input alphabet, 242
input domain, 12, 106, 112, 116, 121
astronomical, 62
complexity, 108
derivation, 108, 153
identification, 123, 125
overlap, 114
partition, 112, 140
size, 108
of sort, 108
source of, 108, 153
subdivision, 110
true, 125
input generator, 27, 28
input space, 12, 297
invalid, 50
modeling, 303
valid, 51
input variable, interaction, 296
inspection team, 92
installation testing, 50
integration
big-bang, 631
bottom up, 631–632
choosing a strategy, 633–634
hierarchy, 620
sandwich, 632–633
sequence, 637
top down, 631
integration errors, 621
integration test adequacy, 664
integration testing, 619–668
cost, 658
goal, 620
integration tests, 431
inTegToolset, 667
Intel Core 2 duo, 661
Intel processor, 52
interaction fault, 305
example, 305
higher level, 308
triggered, 305

interface mutation, 443, 622
interface testing, 46, 48
intermediate node, 172
intermediate result, 160
IPO
  example, 337–342
  horizontal growth, 334
  vertical growth, 334
IPO algorithm, 342
IPO: In Parameter Order, 342
IUT
  apply reset input, 226
  bringing to start state, 226
  extra state, 246
  non-conformance, 225
  number of states, same as in specification, 249
  reset, 226
  testing against control specification, 226
  testing problem, 228
IUT: Implementation Under Test, 215

J
J2EE: Java 2 Enterprise Edition, 54
μJava, *see* MuJava
Java, 31, 53, 66, 74, 118, 364, 398, 516–526
  applet, 346
  condition evaluation, 398
  inheritence mechanism, 518
JavaMetrics, 66
JBlanket, 431
Jester, 517
JMeter, 7, 66
jMock, 668
joke, 68
JRunner, 668
JSystem, 668
Jumble, 533, 534
JUnit, 288, 431, 434, 532, 533, 604, 605–609, 611, 613
  @Before, 609
  @BeforeClass, 609
  @Test, 616
  annotation, 609
  setup(), 605, 609, 617
  tearDown, 605
  tearDown(), 618

K
k-distinguishable, 212, 223
k-dr analysis, 411, 440
k-dr chain, 411, 419–421
  coverage example, 419–421
k-dr coverage, 421
k-dr interaction, 411, 421
Kemerer, Chris F., 32
k-equivalence, 223, 235–238
k-equivalence partition, 235
  convergence of construction process, 238
  example, 235–238
  uniqueness, 289
keyboard, 131
kill process signal, 276
KLOC: Kilo, or thousand, Lines of Code, 29
Klockwork, 91
Knuth, Donald, 539
Kripke structures, 45
  label
    edge, 219
    multiple, 219

L
lamp, 216
language, 99
  catenation, 99
  finite, 99
  procedural, 365
LAPSE: Lightweight Analysis for Program Security in Eclipse, 91
Latin alphabet, 308
Latin square, 308–310
  example, 308–310
  isomorphic, 309
  mutually orthogonal, 308
Lava, 516, 528, 534
lazy evaluation in C, 398
LCSAJ: Linear Code Sequence and Jump, 378–382
  adequate test error revealed, 403
  example, 379–380
  exercised, 379
  traversed, 379
LCSJ criterion, 426
LDRA Testbed, 100, 434
leaf node, 73, 183
leftmost digit, 141
left rotation, 507
Leonhard Euler, 311
Level C software, 533
level number, 638–639
Linux, 148, 345
liquid container, 178
literal, 73
live definitions, 412
LoadRunner, 7
load testing, 52, 54
logical context negation, 494

logical operators: and , or , xor, 371
login, 552
loop, 80
  infinite, 556
  paths, 81
LOTOS, 224

M
Mac, 320
machine equivalence, 223
Mac OS, 297, 324
maintenance, 48
major level, 637–638
Maldonado, José, 478, 534
Malicious Programmer Hypothesis, 537
Marathon, 7
matrix
  inversion, 23
  multiplication, 26
MC adequacy, 384
McCabe metric, 100
McCabe, Thomas, 30
MC/DC, 70, 533
MC/DC adequacy, 368, 376–379
  error detection, 397
  error not detected, 397
  error not revealed, 403
  minimal tests, 396
  size, 387
MC/DC coverage, 100, 369, 383, 387–395
  example, 383, 390
  MC part, 389
MC/DC coverage criterion, 385, 387, 390, 425
MC/DC criterion, 62, 70, 383, 396, 426, 429
  error detection, 437
MC/DC testing, 474
MC: Multiple condition, 382
Mealy machine, 218
measurement, 28
medical device, 332, 552
memory constraint, how to overcome, 433
memory leak, 35
menu driven application, 15
menu pulldown, 114
method
  in Java, 538
  overloading, 523
method coverage, 364, 433
  in context, 364
metric, 28
  complexity, 39
  dynamic, 33
  Halstead, 30
  McCabe, 100
  organizational, 28–29
  process, 30
  product, 30–32
  product quality, 29–30
  project, 29–30
  for project monitoring, 33
  software, 635
  software science, 31, 69
  static, 33
Meyers, Bertrand, 63
microprocessor, 36
microprocessor-memory bus, 433
Microsoft, 287, 288, 434, 595
microwave oven, example, 227
MIDI: Musical Instrument Digital Interface, 224
milk, 112, 113
Miller, Keith, 541
Milu, 532, 534
minimal covering set, 588, 600
minimality of FSM, 259
minimal machine, 223
minor level, 638, 642–646
Miranda, 543
missing initialization, 539–540
missing path error, 91
missing state error, 229–230, 285
mistakes
  common programming, 470
  by C programmers, 470
  by Java programmers, 469
  overloading related, 523
mixed level covering array
  example, 337–342
  generation, 333–334
mixed level covering arrays, 330–332
  generation procedure, 333–342
mixed level orthogonal array, example, 328
mock
  nice, 615
  object, 610, 617
  strict, 615
model-based testing, 44, 45, 47, 68
model based testing, 642
model checker, 44
model checking, 44, 45, 65
model, finite state, 45
ModelJUnit, 288
moderator, 43
modification traversing tests, 545, 554, 557, 567, 571, 586

module prioritization for inspection, 43
MOLS
  complete set, 310
  existence, 311
  maximum number, 344
  procedure for construction, 311–312
  shortcomings in the generation of test configurations, 325
MOLS: Mutually Orthogonal Latin Squares, 310
monitor, 164
Moore machine, 218
Morel, Larry, 541
Mothra, 534
Mozilla, 320
Mozilla Foundation, 67
MPH: Malicious Programmer Hypothesis, 537
MsGAT, 534
MuJava, 478, 516, 532, 534
multidimensional partitioning, 121, 122, 123, 148
  exponential increase, 123
multiple condition coverage, 375, 376, 382
  example, 375, 376
multiple indirection operator, 496
multiple UIO sequences, 264
multiplexer, 35
musical instruments, 224
mutagenic operator, *see* mutation operator
mutant
  analysis, 471
  behavior same as parent, 450
  classification, 457
  compiling, 471
  conditions for distinguishing, 463
  considered distinguished, 463
  distinghuishing conditions example, 464
  distinguished, 231, 449, 456, 464
  drastic, 448
  equivalent, 459, 463
  error hinting, 478
  error revealing, 478
  execution, 457, 471
  extinguished, 463
  first order, 445, 446, 473
  first order, count, 231
  of FSM, 231, 289
  of FSM, example, 231
  generation, 455
  higher order, 445, 446, 534
  higher order example, 445
  killed, 463
  live, 456, 459, 460
  no change in behavior, 448
  non-equivalent, 473
  n^{th} order, 445
  order of execution, 537
  parent, 449
  reliability indicating, 478
  second order, 445, 473
  smallest number, 536
  syntactically correct, 466
  syntactically correct program, 444
  syntactic validity, 491
  terminated, 463
  third order, 445
mutant equivalence
  under strong mutation, 474
  undecidable problem, 474
  under weak mutation, 474
mutant generation, 455
mutant operator, *see* mutation operator
mutation, 444
  access to source code, 443
  act of slightly changing a program, 444
  analysis, 443
  arrays, 511, 512, 514
  as black-box technique, 443
  break statement, 502
  condition, 499
  constant, 467
  constrained, 531
  continue, 502, 506
  continue by break, 502
  cost of, 529
  of C programs, 444
  of design, 532
  discovery of subtle flaws, 451–452
  domain traps, 514
  do-while statement, 504
  entities not mutated, 479
  example, 444
  first order, 534
  of Fortran programs, 443
  goto label, 502
  higher order, 534
  high level languages, 443
  interface, 48, 59, 443, 533, 622
  of Java programs, 443
  linearization, 480, 481
    example, 480
  list of tools, 534
  move brace, 507
  multiple trip, 505
  myths, 541

operator, 467
pointers, 513
program constants, 489
reduced linearized sequence, 480, 481
scalability, 446
score, 452
semantic change, 446
sequence operator, 506
of sqrt to sqr, 540
statement, 501
statement deletion, 499
statement mutation, 468
strong, 449
structures, 513
structures, example, 513
switch statement, 509
systematic procedure for scenario generation, 451
technique for test assessment and enhancement, 443
testing, 443
trap statement execution, 498
twiddle, 515
variable, 468, 511
weak, 449
what is not mutated, 479
while statement, 504
as white-box technique, 443
why?, 450
mutation operator
count of, 525
mutation operator, 465
ABS, 465, 466
accessor modifier example, 525
AMC, 469
binary, 491
bitwise negation, 494
cast operator replacement, 496
casts not mutated, 496
categories, 467
CCCR example, 490
CCR, 466
class, 516
class related for Java, 517
comparable replacement, 491
complete set for C, 478
CRCR, 539
CRCR example, 490
dependence on language, 468
design, 469
design guidelines, 469–470
domain and range, 491
domain of variable, 468
for dynamic binding, 521
effectiveness, 471
generative device, 465
ideal set, 470, 471, 537
incomparable replacement, 491
increment and decrement, 492
indirect operator precedence, 495
inheritence, 518
IOP, 538
for Java, 469, 538
Java specific, 524
language specific, 467
logical negation, 493
manufacturing variables, 468
method overloading, 523
method overloading example, 523
minimality, 472
OAN, 541
OCOR example, 496
for polymorphism, 521
poor naming choice, 537
precise definition, 472
representativeness, 472
ROR, 466
simple mistakes, 467
SMTC example, 506
SMTT example, 505
SMVB example, 508
SRSR example, 501
SSOM example, 507
SSWM example, 509
STRI example, 499
STRP, 467
STRP example, 498
subset, 530
traditional for Java, 517
unary operator, 492
VDTR for C, 467
VLSR in C, 475
VTWD example, 516
mutation score
0, 461
1, 461
computation of, 461
cost of high, 536
golden, 462
mutation testing, 35
within budget, 528
class prioritization, 529
cost, 471
incremental, 462
for novice, 535
tools, 533
mutually singular, 73

N
NAND gate, 36
NASA, 623
NDFSM, 219
n-distinguishable, 284
Netscape, 320
network connection, 296
networked environment, 151
network server, 226
NFA, 219
NFSM, 219
NIST, 342
node, 406
  data, 95
  descendent, 80
  End, 80, 93, 94
  internal, 73
  predecessor, 102
  predicate, 95, 96
  reference, 560
  Start, 80, 93
  successor, 80
Node type, 173
non-core input, 260
non-deterministic FSM, 218
non-embedded application, 583
non-embedded systems, 431
non-OO languages, 630
non-singular expression conflicts, 211
NOT-node, 73
NP-complete integration order, 637
n-switch, 284
n-switch cover, 285, 293
nuclear plant, 447
nuclear plant control, 38
null output, 260
numerical program, 541
NUnit, 605

O
object coverage, 433
Object, mock, 609
Object Relation Diagram, 623
observation
  external, 450
  internal, 450
  points of, 449
O constraint, 162
ODC: Orthogonal Defect Classification, 39
office tools, 431
OFF point, 155, 158, 208
ON-BRIGHT, 216
onto set product, 183
OO metrics, 32
OO programs, 630
OO testing, 53
open systems, 431
operating system, 297
operational correctness, 50
operational profile, 10, 20, 32, 61
operation error, 229
operator mutation, 468
oracle, 25–28, 54, 65
  automation, 69
  construction, 26
  construction using effects, 197
  example, 27
  human, 26
ORD, 623, 624, 626–627, 637–638, 646, 658, 667, 668
  density, 659
  precise, 628
  randomly generated, 659
ordered data context, 412, 423
  coverage, 423
  covered, 423
ordered data context coverage, 440
ordered elementary data context, 412, 423
  covered, 423
  example, 414
OR-node, 73, 173–175
orthogonal arrays, 303, 326
  alternate notation, 327
  design, 342
  example, 326
  index of, 326
  mixed, 323
  mixed level, 326
  for a pacemaker, 322
  run of, 326
  strength of mixed level, 327
Orthogonal Defect Classification, 38
OS bridge, 302
output table, 236
output conditions, 160
output function, 219, 234
Oxygen, 465

P
pacemaker, 552
  orthogonal array, 333
pairwise coverage, 313
pairwise design, 313
  construction using MOLS, 318
  multivalued factors, 318
  procedure for generation, 314

pairwise interaction fault, 306
due to missing condition, 306
revealed, 306
pairwise testing, 46, 296
parameter, same as factor, 333
PaRTeG, 534
partial-W method, 215
Partition testing, 46, 147
path, 93
(, 77
), 83
basis, 83, 84
basis set, 84
complete, 75
condition, 84
data flow, 293
deciding infeasibility, 474
def-clear, 408
definition clear, 410
distinct, 83, 101
domain, 85, 87
border segment, 87
enumeration, 435
execution, 93
exponential, 81
infeasible, 79, 358, 365, 422
invalid, 80
length, 83, 101
linearly independent, 83
loops, 82
number of, 81, 356
number example, 81
prime, 438
round trip, 438
shortest, 274
simple, 438
subpath, 79
traversal, 412
tricky, 363
vector, 79, 101
path condition, non-lionear, 208
path coverage criterion, 351, 356, 360
paths, 351
path selection error, 91
payroll processing, 54, 108, 117
PC, 320
PDG, *see* program dependence graph
PDom, *see* post-dominator
PDS
input space, 298
PDS: Pizza Delivery Service, 298
penetration testing, 51
Pentium, 302
performance, 9, 555
analysis, 594
testing, 50
Perl, 595
Pester, 534
Petri net, 41, 106, 214
PHP, 595
pin code, 551
Pizza, 298
toppings, 298
variety, 298
pizza service
testing, 329
tests, 328
planet name, 148
PMS: Pilot Mutation System, 534
ON point, 155, 208
pointer variable, 404
polymorphism, 626
post-dominator, 93, 94
example, 94
immediate, 94
relation, 94
strict, 94
tree, 94
potential dependence, 575, 579, 599
Pounder, 7
precedence, 73
predicate, 95, 176, 367
compound, 72, 74
converted to Boolean expression, 73
entire, 179
non-singular, 201
simple, 72
predicate constraint, 179
predicate logic, 41
predicate node, 577
predicate test generation
program based, 203
specification based, 200
predicate testing, 176
in practice, 200
prematurely terminating execution sequence, 482
premature termination, 556
price database, 112
prime path, 438
primitive types in C, 526
printer, 114, 164
color laserjet, 114
manufacturer, 113
model, 113
networked, 297
prioritization algorithm, 593

probe test inadequacy, 431
procedural language, 364
procedure call, 76
process control system, 178
process maturity, 67
processor
  Athelon, 302
  Pentium, 302
productivity, 43
product reliability, 32
product stability, 33
product type, 141
product of variables, 73
profiling
  non-intrusive, 433
  program execution, 433
program
  analysis, 35, 76
  behavior, 13, 14
  complexity, 30
  continously running, 226
  correct behavior, 25
  correctness, 37
  crash, 65
  error free, 353
  execution, 23
  execution of sequential, 378
  exponentiation, 355
    coverage domain, 356
    test enhancement, 357
  hang, 65
  initial state, 14
  mutation, 443
  numerical, 541
  older, 364
  partial, 634
  review, 45
  separate runs, 364
  single execution, 363
  state, 13
  state diagram, 13
  statement, 95
  termination of execution, 378
  under test, 25
  untested parts, 360
  variable, 87, 93
  verification, 37
program based tests, 200
program dependence graph, 95, 97, 101, 571, 573, 579, 599
  dynamic, 573
  example, 97
program differencing, 596
program graph, *see* control flow graph
programmer, 5
programming mistakes, 517
program mutants, 463
program mutation, 443 *see* mutation
Program, OO, 630
program revalidation, 548
program size, 31
program specification, 147
prohibited pairs, 346
property, 144
property list, 144
Proteum, 478, 534
Proteum/IM, 534
protocol, 106, 216
psychology in usability testing, 9
punched cards, 131
Purdue University, 478
Purify, 91
p-use, 407, 410, 417
  adequacy, 428
  adequacy does not imply c–use adequacy, 428
  coverage, 417, 427
  coverage example, 418
  feasible, 438
  infeasible, 422
  minimization, 412
  number of, 415
p-use: predicate use, 405
Python, 534, 595, 616
PyUnit, 532

Q
quality
  software, 8
  usability, 9
quality attribute, 8
  completeness, 9
  consistency, 9
  dynamic, 9
  performance, 9
  static, 8
quasigroup, 343

R
radiation leak, 4
RAM, 164
random request generator, 34
random testing, 7, 64
range
  implicit specification, 117
  multiple, 118
  specification, 119
ranking, requirements, 559

RAP set, 498
R constraint, 162
RDDG: Reduced Dynamic Dependence Graph, 582
reachability, 463
  condition, 463
  example, 476
reaching definition, 577
reactor monitoring system, 447
reader, 43
real-time constraints, 433
real-time software, 53, 54
real-time system, 106, 229
real-time testing, 53
recorder, 43
record, student, 54, 120
reference
  global scalar, 486
  global set example, 486
  local scalar, 486
  local set example, 486
  multilevel, 486
  scalar, 486
  sets, 486
reflexive, 289
regress, 548
regression testing, 41, 93, 548–596, 627
  commercial tools, 594
  corrective, 548
  handling declarations, 567
  obsolete tests, 556
  problem, 555, 556
  progressive, 548
  random selection, *see* random testing
  revalidation, 548
  safe, 558
  system testing phase, 595
  test all, 557
  test minimization, 550
  test prioritization, 559
  test process, 549
  test revalidation, 550
  test selection, 550
  using integer programming, 558
regression test selection problem, 550, 555
regular expression, 98, 99
regulatory organization, 447
relation, 111
  depends, 627
  inheritance, 625
  static, 626
relational expression, 72
relational operator, 72
  testing, 207
relational operator fault, 177, 210
release decision, 30
relevant slice, 579
reliability, 10, 19
  assesment, 50
  estimated, 60
  estimation, 20
  high, 296
  testing, 50
  true, 59
reliability estimation, 7
relop: relational operator, 72
repeatability, 330
requirement, independence of conditions, 172
requirements, 11, 40
  ambiguous, 401
  coverage, 430
  familiarity, 110
  functional, 51, 350
  as ideas, 106
  incomplete, 401
  incorrect, 401
  informal, 106
  non-functional, 350
  performance, 51
  revealing ambiguities, 400
  rigorous, 106
  rigorously specified, 106, 153
  unambigous specification, 111
requirements testing, 53
reset input, 226, 260
residual coverage, 590, 591
resistance, 50
retest, 635
retest edge, 638
retesting
  cost, 635, 637
  cost reduction, 637
review plan, 42
right associativity, 73
RLS: Reduced Linearized Sequence, 480, 481
robustness, 51, 666
robustness testing, 51, 52
role
  developer, 7
  tester, 7
root node, of predicate, 199
ROR operator, 527, 532, 536
RTS: Regression Test Selection, 551
Ruby, 596
run, 297

S
Safari, 298
safe regression test selection, 558
saturation effect, 59, 63, 65, 68, 70
  impact on test process, 63
saturation region, 60
SCC, 654
  non-trivial, 654
SCC: Strongly Connected Components, 650
scenario, 203
  ATM, 212
  combining, 212
  events, 203
  from statechart, 207
  test case, 206
  from UML diagram, 207
  from Z, 207
SDL, 224
search mode, 27
secure connection, 208
selective mutation, 533
selector expression, 144, 146
Selenium, 594
  IDE, 595
  WebDriver, 595
semantic change, 446
Semantic Designs, 66
semantic relations, 131
semicolon (;), 69
sequence diagram, 41, 214
server class, 635
set cover
  naive algorithm, 600
  optimization problem, 587
set cover problem, 587
  greedy algorithm, 587
  naive algorithm, 587
  tricky, 600
SET interface, 543
SETL, 543
set product, 124, 183
set type, 543
setUp(), 605
short circuit evaluation, 397, 398
shutdown system, 447
side effects, 226
signature, 261
  instead of UIO sequence, 261
SilkTest, 595
simple change, 445
Simple compound, 72
simple condition infeasible, 371
simple path, 438
simple programming mistakes, 467
simple relational expression, 179
simulated environment, 125
simulator, 552
singular component, 73
skeletal system, 635
slicing, 572
  dynamic, 570
  static, 572
SLOC: Source Lines of Code, 66
smart card, 38
social security, 118
software
  catastrophic, 533
Software Engineering, 531
Software Engineering Terminology, 62
software lifecycle, 48
software quality, 8
software reliability, 10
software systems, 296
software testing, 48
  foundation, 48
sort, 23, 25, 68, 108, 364
  factors, 298
  test generation using factors, 305
source code analysis, 39
source document, 40
SourceForge, 616
SpartanUnit, 616
Spec Explorer, 287
specification, 41
  rewrite, 145
specification based tests, 200
specific stub, number of, 660
spiral model, 55
spiral testing, 55, 57
Spurgeon, John, 616
square root program, 439
stable code, 533
stand-alone item, 164
Start node, 407
state
  accepting, 218
  ON-BRIGHT, 220
  current, 216
  diagram, 216, 217, 227
  diagram of DIGDEC, 217
  ON-DIM, 216, 219
  distinguishable, 223
  of environment, 140
  equivalence, 289
  equivalent, 222
  estimating maximum number of, 243
  example, 14
  final, 14, 218

identification set, 241, 250
example, 242
infection, 463, 476
infection condition, 464, 465
infection example, 476
initial, 14, 218, 562
invalid, 305
k-distinguishable, 223, 289
k-equivalent, 223, 235
of a lamp, 216
of object, 140
OFF, 216, 217
propagation, 463
propagation condition, 464
propagation example, 476
removal, 233
sequence, 14
tabular, 221
transition function, 218, 234
V-equivalence, 289
statechart, 26, 41, 46, 215, 532
UML, 214
Statechart testing, 46
state cover, 282
state cover set, 250
statement, 95
declarative, 364
executable, 364
unreachable, 365, 366
statement coverage, 365
domain, 365
example, 365
statement deletion operator, 537
state space, perturbation, 473
state transition table, 240
state vector, 13
state verification, 609
static, 524
static analysis, 29, 628
example, 91
static analysis tool, 91
static reaching definition, 577
static slice, 572
static testing, 42
stress testing, 35, 50, 51
strict mock, 615
string, 98, 99
alphanumeric, 69
language, 99
non-empty, 136
test input, 98
string concatenation partial, 255
strong conformance, example, 278
strongly connected components, 643
strong mutation, 450, 537
Structural testing, 46, 429, 431
part of functional testing, 431
structured program, 69
structures, 120
stub, 609, 611, 631–638, 643, 646, 656
cost, 634
minimization, 636
not needed, 628
number of, 632, 636
to or not to, 635
server class, 635
smallest number, 659
specific, 635, 636, 654
stubbed, 630
stubbing, 607, 620
stuck-at fault model, 35
student, 120
student record, 54
sub-domain, 18, 296
subgraph, 671
subpath, 79, 438
initial, 79
sub-path, 80
subsumes relation, 427, 428
for c– and p–uses, 441
example, 428
switch
lamp, 216
rotary, 216
switch cover, 282
switch cover criterion, 282
SWS: Software Specification, 623
symbol
input, 218
output, 218
symmetric, 289
syntactic category, 480
syntax, carrier of semantics, 446
Syntax testing, 46
syntax tree, 563, 564, 596
abstract, 73
equivalence, 564
synthesizer, 224
system
integration, 49
testing, 49
system test, 69, 70, 215

T
t, 179
table lamp, 216, 219
Tarjan's algorithm, 643

tasks
  conjoining of, 201
TDD: Test Driven Development, 55, 617
TD: Tai-Daniel's method, 637
tearDown, 605
Telcordia Technologies, 342, 435
telephone network, 24
temporal logic, 44
tennis coach, 5
terminology overlap, 52
test
  adequacy, 350
  adequacy criterion, 351
  adequacy measurement, 353
  color coded, 607
  completeness, 64, 435
  coverage, 351
  data, 21
  data base, 537
  driver, 147
  efficiency, 37
  enhancement, 431
  error revealing, 198
  fault revealing, 198
  frame, 140, 144, 145
  generation, 35
  minimal, 436
  modification, 41
  order, 630
  parameters, 297
  point intermediate, 160
  point internal, 160
  redundant, 463
  representative, 111
  script, 140, 145
  sequence, 637
  specification, 140
  successful, 419
  suite, 37
  team, 106
  time, 396
  timed sequence, 287
  trace back, 400
  value, 297
  vector, 562
  verification, 555
testability
  dynamic metric, 33
  requirement, 34
  static metric, 33
testable code, 8
testable entity, 589
testable unit, 142
test abortion, 594
test adequacy, 415
  alternate procedures using mutation, 462
  based on c–use and p–use, 405
  concurrent assessment, 463
  condition coverage, 433
  k-dr coverage, 410, 411
  MC coverage, 382
  path coverage, 353, 354
  of unit tests, 602
test adequacy assessment, multi-tester version using mutation, 462
test adequacy criteria, 360
test assessment
  foundations, 434
  using mutation, 453
test automation, 7
test automation tools, 7
test box, 133
test case, 21, 47
  documentation, 400
  from factor combination, 297
  fuss about, 364
  new, 360
  redundant, 400, 537
  satisfies constraint, 179
test configuration, 296, 297
  from combinatorial object, 303
test construction-enhancement cycle, 354
test coverage, 36
test data, 21
test data management, 66
test design, 303, 603
test driven development, 603, 605, 617
tested, 351
test effort, 59
  wasted, 61
test enhancement, 353
  example, 353
  foundations, 434
  incremental, 462
  procedure, 354
  process, 354
  purpose, 360
test environment, 60
tester, 7, 142
  adamant, 401
  experience, 110
  frustration, 360

test execution, 6
error prone, 7
manual, 555
test frame, 145
automatic generator, 145
deriving a test case, 146
generation, 146
key, 146
not a test case, 146
redundant, 145, 146
sample, 144
test frame generator, 140
test generation, 621
algorithm, 41
automata theoretic, 282
black box, 62
category partition method, 140
code based, 41
control flow based, 282
from decision table, 175
from EFSM, 288
from finite state machine, 215
for GUI using factors, 299
informal, 40
MC/DC adequate, 383–387
model-based, 44
procedure, 215
for sort using factors, 299
specification based, 58
from statechart, 216
UIO method, 274–275
using heuristics, 168
test harness, 23
testing, 4, 18, 21
acceptability, 49
of actions, 4
ad-hoc, 47
agile, 55, 57
balanced design not needed, 331
beta, 49
black-box, 45, 47, 48
business process, 66
complete, 350, 435
confidence, 38
conformance, 224
data flow, 438
distributed, 66
domain, 89, 90, 154
dynamic, 44
exhaustive, 108, 109
fault-injection, 443
function, 120
functional, 51, 65, 429
glass-box, 429
goal, 5
goal directed, 46
good enough, 350
GUI, 58
human intensive, 7
incremental, 637
integration, 48, 620, 624, 658
interaction, 296
interface, 48
a joke, 68
load, 52, 66
model-based, 44, 45
mutation, 48, 443, 449
mutation based, 481
object, 120
OO, 53
pairwise, 46, 48, 296
partition, 147
penetration, 51
performance, 51
predicate, 176
principles, 63
of product, 4
random, 48, 64
regression, 49, 627
robustness, 51
security, 51
sort, 5
spiral, 55
static, 42, 91
stress, 51
structural, 429
system, 49
thorough, 142, 350
of thoughts, 4
top down, 632
types, 45
unit, 48, 541, 601
user-centric, 9
using W-method, 246
using Wp-method, phase, 1, 252
using Wp-method, phase, 2, 252
and verification, 37
VLSI, 36
in V model, 55
vulnerability, 51
in waterfall development process, 54
web services, 58
white-box, 47, 49, 429
testing problem, 228
testing techniques
classification, 45
testing tree, 243, 244, 286
example, 244

test minimization, 550, 558, 586, 590, 596
coverage based, 600
example, 586
procedure, 588
as set cover problem, 587
Testopia, 67
test optimization, 277
test order, 637
time to generate, 661
test plan, 21
test pool, 24
test prioritization, 559, 590
tests
minimal, 135
modification traversing, 557
obsolete, 555
program based, 200
redundant, 555
regression, 555
specification based, 200
thoroughly designed, 38
test scenario, 68
test script, 54
test selection
boiler control, 129
equivalence class, 128–131
example, 131
test selection tool, 558
test sequence
relative lengths, 287
upper bound on length, 291
test sequencing, 552, 554
test sequencing constraint, 600
test set, 21, 106
BOR adequate, 210
BRE adequate, 210
construction using W-method, 242
decision adequate, 435, 538
effciency, 37
effectiveness, 37
hopelessly inadequate, 461
inadequacy, 353
inadequate with respect to mutation, 453
MC adequate, 436
MC/DC adequate, 382, 436, 437, 439
minimal, 438
minimization, 541
multiple-condition adequate, 435
satisfying BOR, 181
satisfying BRE, 181
satisfying BRO, 181
union, 201
test set efficiency, 37
test set up, 551
test specification, 140
test suite, 21
efficiency, 37
number of tests, 28
test tools, 66, 412
TestTube, 7, 594, 595
T_EX, 539
text processing, 26
t-factor fault, 306
this, 524
time complexity, 600
timed automata, 288
timed input automata, 106
timing, 44
timing constraints, 106, 109
timing properties, 44
TJJM method, 649
TJJM: Traon-Jéeron-Jéezéequel-Morel method, 637
tomato, 112, 113
tour, 275
toy, 216
trace back to requirements, 400
trace, *see* execution trace
transaction flow testing, 53
transfer error, 229, 247, 279
not revealed, 284
transfer sequence, 274
transition cover, 282
transition cover set, 243, 290
transition diagram, 290
transition function, 218
transition tour, 287, 292
transition tour method, 287
transitive, 289
transitive closure, 628
traversal index, 650
triangle
classification, 604
types, 604
true, 72
truth value, 177
TSL, 144, 145
different terminology, 146
specification, 146
syntax, 145
TSL: Test Specification Language, 145
t-tuple, 326
T-UPPAAL, 288
t-way fault vector, 307
t-way interaction fault, 306

U
UCM: Unified Change Management, 67
UIO method, 277
  assumptions, 257
  failure to detect faults, 281
  fault detection example, 278
UIO sequence, 257
  algorithm for generation of, 262–264
  checking, 258
  does not exist, 259
  example, 260, 265–268, 271–273
  length, 257
UIO: Unique Input/Output
    Sequence, 257
UIOv method, 286
UML, 41, 106, 207
  sequence diagram, 214
  statechart, 214
unbalanced design, 331
undefined, 14
unidimensional partitioning, 121, 160
United States, 533, 551
unit, test adequacy, 602
unit testing, 48, 541, 601
  using JUnit, 605
universe defining entities, 479
University of London, 516
Unix, 11, 415
UNIX, 14, 123
usability testing, 9
USB: Universal Serial Bus, 7
use
  global, 406
  input variable, 412
  local, 406
use case, 106
use case map, 206
user action, 206
user centric testing, 9
user-defined, scenario, 205
uses, 404
use set, 407

V
validity check, 136
variable
  global set, 487
  live definition, 412
  local set, 488
variable negation, 73
variable redefinition, 91
VBScript, 66
vector of factor levels, 307
V-equivalence, 222
verification, 37
  behavior, 609
  state, 609
video cassette, 27
Virtual PC, 302
VLSI, 35
V-model, 55
Voas, Jeffery, 541
vulnerability testing, 51, 52

W
walkthroughs, 42
WAPT, 7
Wayne State University, 320
weak conformance, 278, 292
weak mutation, 449, 537
web based company, 164
web service, 51, 58, 551, 554
  testing, 51, 58
WebSphere Studio, 54
weekly salary, 108
while statement, 95
white-box
  category, 48
  criteria, 665
  test adequacy criteria, 351
  testing, 47, 429
  tests, 203
Windows operating system, 38
Windows XP, 296
WinRunner, 7, 66
withdraw, 552
W-method, 215, 242, 287, 292
  assumptions, 242
  error detection, 249
  fault detection effectiveness, 282
  upper bound on number of tests, 291
Wp-method, 242, 287
  example, 252, 253, 255
  fault detection effectiveness, 282
  partial W method, 250
  procedure, 251
W procedure, 239–240
writing pen, 148
W set
  example, 234
  state characterization set, 234

X
X.25, 224
XOR-node, 73, 183
XP: eXtreme Programming, 55

x: set product, 200
XTest, 7
X-testing, 45

Y
yogurt, 112

Z
Z, 47, 532, 543
zip code, 298, 551
Z notation, 58, 70
Z specification, 70
χSUDS, 412, 435, 594

CSE 4017	Software Testing	CT C LT 3
Prerequisite:	Software Engineering	

Objectives:

- **Apply the fundamental knowledge of testingtechniques appropriate to the problems**
- **To provide an overview of finding bugs**
- **To understand and develop well-structuredtesting techniques.**

Expected Outcomes:
At the end of the course, students should able to

- Test one problems/ computation using various testing techniques
- Remove errors and bugs from a program.

Student Outcomes (SO):
a. An ability to apply the knowledge of mathematics, science and computing appropriate to the discipline
b. An ability to analyze a problem, identify and define the computing requirementsappropriate to its solution.
c. An ability to design, implement and evaluate a system / computer-based system,process, component or program to meet desired needs
l. An ability to apply mathematical foundations, algorithmic principles and computer science theory in the modelling and design of computer-based systems (CS)

Unit No	Unit Content	No. of hours	SOs
1	**Overview of Testing:** Software Testing Definition, Debugging, Testing vs debugging.Purpose of testing, Dichotomies, model for testing, consequences of bugs, taxonomy of Bugs.	4	a,b, c
2	**Flow graphs and Path testing:** Basics concepts of path testing, predicates, path predicates and achievable paths, path sensitizing, path instrumentation, application of path testing. **Transaction Flow Testing:** Transaction flows, transaction flow testing techniques.	6	a, b, c
3	**Dataflow testing**:-Basics of dataflow testing, strategies in dataflow testing, application of dataflow testing. **Domain Testing**:-domains and paths, Nice & ugly domains, domain testing, domains and interfaces testing, domain and interface testing, domains and testability.	6	a, b, c
4	**Paths, Path products and Regular expressions: P**ath products & path expression, reduction procedure, applications, regular expressions & flow anomaly detection.Enumerated data type – Union.	6	a, b, c
5	**Logic Based Testing:** Overview, decision tables, path	6	a, b, c

	expressions, kv charts, specifications.		
6	**Guest Lecture on Contemporary Topics**	**2**	
	Total Lecture:	**30**	

Mode of Teaching and Learning: Flipped Class Room, One Lecture to be videotaped, Digital/Computer based models to augment lecture for practice/tutorial, 2 hours lectures by industry experts on contemporary topics

Mode of Evaluation and assessment:
The assessment and evaluation components may consist of unannounced open book examinations, quizzes, student's portfolio generation and assessment, and any other innovative assessment practices followed by faculty, in addition to the Continuous Assessment Tests and Final Examinations.

Text Books:

1.	Software Testing techniques - BarisBeizer, Dreamtech, second edition.
2.	Software Testing Tools – Dr.K.V.K.K.Prasad, Dreamtech.

Reference Books:

1.	Art of Software Testing – Meyers, John Wiley.
2.	The craft of software testing - Brian Marick, Pearson Education.

Recommendation by the Board of Studies on	18.06.2020
Approval by Academic council on	
Compiled by	Dr. Sandip Mal

CONTENTS

1	**Unit-I**	**Page No**
1.1	Introduction: Purpose of Testing	6
1.2	Dichotomies 8-10	8
1.3	Model for Testing	11
1.4	Consequences of Bugs	12
1.5	Taxonomy of Bugs	18
1.6	Flow graphs and Path testing: Basics Concepts of Path Testing	19
1.7	Path Predicates and Achievable Paths	24
1.8	Path Sensitizing	25
1.9	Path Instrumentation	26
1.10	Application of Path Testing	31
1.11	Part A Questions	40
1.12	Part B Questions	41
2	**Unit-II**	
2.1	Transaction Flow Testing: Transaction Flows	42
2.2	Transaction Flow Testing Techniques	44
2.3	Dataflow testing: Basics of Dataflow Testing	45
2.4	Strategies in Dataflow Testing	52
2.5	Application of Dataflow Testing	54
2.6	Part A Questions	58
2.7	Part B Question	58
3	**Unit-III**	
3.1	Domain Testing: Domains and Paths	60
3.2	Nice & Ugly Domains	62
3.3	Domain testing	64
3.4	Domains and Interfaces Testing	67
3.5	Domain and Interface Testing	69
3.6	Domains and Testability	72
3.7	Part A Questions	78
3.8	Part B Questions	79
4	**Unit-IV**	
4.1	Paths, Path products and Regular expressions: Path Products & Path Expression	80
4.2	Reduction Procedure	83
4.3	Applications	86
4.4	Regular Expressions & Flow Anomaly Detection	96
4.5	Logic Based Testing: Overview	103
4.6	Decision Tables	104
4.7	Path Expressions	108
4.8	KV Charts, Specifications	114
4.9	Part A Questions	116
4.10	Part B Questions	117
5	**Unit-V**	
5.1	State, State Graphs and Transition Testing: State Graphs	118
5.2	Good& Bad State Graphs	120
5.3	State Testing	121
5.4	Testability Tips	122
5.5	Graph Matrices and Application: Motivational Overview	124

5.6	Matrix of Graph	127
5.7	Relations	128
5.8	Power of a Matrix	128
5.9	Node Reduction Algorithm	129
5.10	Building Tools	129
5.11	Part A Questions	130
5.12	Part B Questions	131

UNIT- I

INTRODUCTION

What is testing?

Testing is the process of exercising or evaluating a system or system components by manual or automated means to verify that it satisfies specified requirements.

The Purpose of Testing

Testing consumes at least half of the time and work required to produce a functional program.

- MYTH: Good programmers write code without bugs. (It'swrong!!!)
- History says that even well written programs still have 1-3 bugs per hundredstatements.

Productivity and Quality in Software:

- In production of consumer goods and other products, every manufacturing stage is subjected to quality control and testing from component to finalstage.
- If flaws are discovered at any stage, the product is either discarded or cycled back for rework andcorrection.
- Productivity is measured by the sum of the costs of the material, the rework, and the discarded components, and the cost of quality assurance andtesting.
- There is a tradeoff between quality assurance costs and manufacturing costs: If sufficient time is not spent in quality assurance, the reject rate will be high and so will be the net cost. If inspection is good and all errors are caught as they occur, inspection costs will dominate, and again the net cost willsuffer.
- Testing and Quality assurance costs for 'manufactured' items can be as low as 2% in consumer products or as high as 80% in products such as space-ships, nuclear reactors, and aircrafts, where failures threaten life. Whereas the manufacturing cost of software is trivial.
- The biggest part of software cost is the cost of bugs: the cost of detecting them, the cost of correcting them, the cost of designing tests that discover them, and the cost ofrunning thosetests.
- For software, quality and productivity are indistinguishable because the cost of a software copy istrivial.
- Testing and Test Design are parts of quality assurance should also focus on bug prevention. A prevented bug is better than a detected and correctedbug.

Phases in a tester's mental life:

Phases in a tester's mental life can be categorized into the following 5 phases:

1. **Phase 0: (Until 1956: Debugging Oriented)** There is no difference between testing and debugging. Phase 0 thinking was the norm in early days of software development till testing emerged as adiscipline.
2. **Phase 1: (1957-1978: Demonstration Oriented)** the purpose of testing here is to show that software works. Highlighted during the late 1970s. This failed because the probability of showing that software works 'decreases' as testing increases. I.e. the more you test, the more likely you will find abug.
3. **Phase 2: (1979-1982: Destruction Oriented)** the purpose of testing is to show that software doesn't work. This also failed because the software will never get released as you will find one bug or the other. Also, a bug corrected may also lead to anotherbug.
4. **Phase 3: (1983-1987: Evaluation Oriented)** the purpose of testing is not to prove anything but to reduce the perceived risk of not working to an acceptable value (Statistical Quality Control). Notion is that testing does improve the product to the extent that testing catches bugs and to the extent that those bugs are fixed. The product is releasedwhenthe confidence on that product is high enough. (Note: This is applied to large software products with millions of code and years ofuse.)
5. **Phase 4: (1988-2000: Prevention Oriented)** Testability is the factor considered here. One reason is to reduce the labor of testing. Other reason is to check the testable and non-testable code. Testable code has fewer bugs than the code that's hard to test. Identifying the testing techniques to test the code is the main keyhere.

Test Design:

We know that the software code must be designed and tested, but many appear to be unaware that tests themselves must be designed and tested. Tests should be properly designed and tested before applying it to the actualcode.

Testing isn't everything:

There are approaches other than testing to create better software. Methods other than testing include:

1. **Inspection Methods:** Methods like walkthroughs, desk checking, formal inspections and code reading appear to be as effective as testing but the bugs caught don't completely overlap.
2. **Design Style:** While designing the software itself, adopting stylistic objectives such as testability, openness and clarity can do much to preventbugs.
3. **Static Analysis Methods:** Includes formal analysis of source code during compilation. In earlier days, it is a routine job of the programmer to do that. Now, the compilers have taken over thatjob.
4. **Languages:** The source language can help reduce certain kinds of bugs. Programmers find new bugs while using newlanguages.

5. **Development Methodologies and Development Environment:** The development process and the environment in which that methodology is embedded can prevent many kinds ofbugs.

Dichotomies:

- **Testing VersusDebugging:**
  Many people consider both as same. Purpose of testing is to show that a program has bugs. The purpose of testing is to find the error or misconception that led to the program's failure and to design and implement the program changes that correct the error.
  Debugging usually follows testing, but they differ as to goals, methods and most important psychology. The below tab le shows few important differences between testing and debugging.

Testing	Debugging
Testing starts with known conditions, uses predefined procedures and has predictable outcomes.	Debugging starts from possibly unknown initial conditions and the end cannot be predicted except statistically.
Testing can and should be planned, designed and scheduled.	Procedure and duration of debugging cannot be so constrained.
Testing is a demonstration of error or apparent correctness.	Debugging is a deductive process.
Testing proves a programmer's failure.	Debugging is the programmer's vindication (Justification).
Testing, as executes, should strive to be predictable, dull, constrained, rigid and inhuman.	Debugging demands intuitive leaps, experimentation andfreedom.
Much testing can be done without design knowledge.	Debugging is impossible without detailed design knowledge.
Testing can often be done by an outsider.	Debugging must be done by an insider.
Much of test execution and design can be automated.	Automated debugging is still a dream.

- **Function versusStructure**:
  - Tests can be designed from a functional or a structural point ofview.
  - In **Functional testing**, the program or system is treated as a black box. It is subjected to inputs, and its outputs are verified for conformance to specified behavior. Functional testing takes the user point of view- bother about functionality and features and not the program'simplementation.

- In **Structural testing** does look at the implementation details. Things such as programming style, control method, source language, database design, and coding details dominate structuraltesting.
- Both Structural and functional tests are useful, both have limitations, and both target different kinds of bugs. Functional tests can detect all bugs but would take infinite time to do so. Structural tests are inherently finite but cannot detect all errors even if completelyexecuted.

- **Designer versusTester**:

  - Test designer is the person who designs the tests where as the tester is the one actually tests the code. During functional testing, the designer and tester are probably different persons. During unit testing, the tester and the programmer merge into oneperson.
  - Tests designed and executed by the software designers are by nature biased towards structural consideration and therefore suffer the limitations of structural testing.

- **Modularity versusEfficiency**:

  A module is a discrete, well-defined, small component of a system. Smaller the modules, difficult to integrate; larger the modules, difficult to understand. Both tests and systems can be modular. Testing can and should likewise be organized into modular components. Small, independent test cases can be designed to test independent modules.

- **Small versusLarge:**

  Programming in large means constructing programs that consists of many components written by many different programmers. Programming in the small is what we do for ourselves in the privacy of our own offices. Qualitative and Quantitative changes occur with size and so must testing methods and quality criteria.

- **Builder versusBuyer:**

  Most software is written and used by the same organization. Unfortunately, this situation is dishonest because it clouds accountability. If there is no separation between builder and buyer, there can be no accountability.
- The different roles / users in a systeminclude:
  1. **Builder:** Who designs the system and is accountable to thebuyer.
  2. **Buyer:** Who pays for the system in the hope of profits from providing services?
  3. **User:** Ultimate beneficiary or victim of the system. The user's interests are also guardedby.
  4. **Tester:** Who is dedicated to the builder'sdestruction?
  5. **Operator:** Who has to live with the builders' mistakes, the buyers' murky (unclear) specifications, testers' oversights and the users'complaints?

MODEL FOR TESTING:

Figure 1.1: A Model for Testing

Above figure is a model of testing process. It includes three models: A model of the environment, a model of the program and a model of the expectedbugs.

- Environment:
  - A Program's environment is the hardware and software required to make it run. For online systems, the environment may include communication lines, other systems, terminals andoperators.
  - The environment also includes all programs that interact with and are used to create the program under test - such as OS, linkage editor, loader, compiler, utility routines.
  - Because the hardware and firmware are stable, it is not smart to blame the environment forbugs.
- Program:
  - Most programs are too complicated to understand indetail.
  - The concept of the program is to be simplified in order to testit.
  - If simple model of the program doesn't explain the unexpected behavior, we may have to modify that model to include more facts and details. And if that fails, we may have to modify theprogram.
- Bugs:
  - Bugs are more insidious (deceiving but harmful) than ever we expect them tobe.
  - An unexpected test result may lead us to change our notion of what a bug is and our model ofbugs.
  - Some optimistic notions that many programmers or testers have about bugs are usually unable to test effectively and unable to justify the dirty tests most programsneed.
  - Optimistic notions about bugs:
    1. **Benign Bug Hypothesis:** The belief that bugs are nice, tame and logical. (Benign: NotDangerous)

2. **Bug Locality Hypothesis:** The belief that a bug discovered with in a component affects only that component'sbehavior.
3. **Control Bug Dominance:** The belief those errors in the control structures (if, switch etc) of programs dominate thebugs.
4. **Code / Data Separation:** The belief that bugs respect the separation of code anddata.
5. **Lingua Salvatore Est.:** The belief that the language syntax and semantics (e.g. Structured Coding, Strong typing, etc) eliminates mostbugs.
6. **Corrections Abide:** The mistaken belief that a corrected bug remains corrected.
7. **Silver Bullets:** The mistaken belief that X (Language, Design method, representation, environment) grants immunity from bugs.
8. **Sadism Suffices:** The common belief (especially by independent tester) that a sadistic streak, low cunning, and intuition are sufficient to eliminate most bugs. Tough bugs need methodology andtechniques.
9. **Angelic Testers:** The belief that testers are better at test design than programmers is at codedesign.

- **Tests:**
  - Tests are formal procedures, Inputs must be prepared, Outcomes should predict, tests should be documented, commands need to be executed, and results are to be observed. All these errors are subjected toerror
  - We do three distinct kinds of testing on a typical software system. Theyare:
    1. **Unit / Component Testing:** A **Unit** is the smallest testable piece of software that can be compiled, assembled, linked, loaded etc. A unit is usually the work of one programmer and consists of several hundred or fewer lines of code. **Unit Testing** is the testing we do to show that the unit does not satisfy its functional specification or that its implementation structure does not match the intended design structure. A **Component** is an integrated aggregate of one or more units. **Component Testing** is the testing we do to show that the component does not satisfy its functional specification or that its implementation structure does not match the intended designstructure.
    2. **Integration Testing: Integration** is the process by which components are aggregated to create larger components. **Integration Testing** is testing done to show that even though the components were individually satisfactory (after passing component testing), checks the combination of components are incorrect orinconsistent.
    3. **System Testing:** A **System** is a big component. **System Testing** is aimed at revealing bugs that cannot be attributed to components. It includes testing for performance, security, accountability, configuration sensitivity, startup andrecovery.
- **Role of Models:** The art of testing consists of creating, selecting, exploring, and revising models. Our ability to go through this process depends on the number of different models we have at hand and their ability to express a program'sbehavior.

CONSEQUENCES OF BUGS:

- **Importance of bugs:** The importance of bugs depends on frequency, correction cost, installation cost, andconsequences.
    1. **Frequency:** How often does that kind of bug occur? Pay more attention to the more frequent bugtypes.
    2. **Correction Cost:** What does it cost to correct the bug after itis found? The cost is the sum of 2 factors: (1) the cost of discovery (2) the cost of correction. These costs go up dramatically later in the development cycle when the bug is discovered. Correction cost also depends on systemsize.
    3. **Installation Cost:** Installation cost depends on the number of installations: small for a single user program but more for distributed systems. Fixing one bug and distributing the fix could exceed the entire system's developmentcost.
    4. **Consequences:** What are the consequences of the bug? Bug consequences can range from mild tocatastrophic.

  A reasonable metric for bug importance is
  Importance= ($) = Frequency * (Correction cost + Installation cost + Consequential cost)

- **Consequences of bugs:** The consequences of a bug can be measure in terms of human rather than machine. Some consequences of a bug on a scale of one to tenare:
    1 **Mild:**Thesymptomsof thebugoffendusaesthetically(gently);amisspelled output or a misalignedprintout.
    2 **Moderate:** Outputs are misleading or redundant. The bug impacts the system's performance.
    3 **Annoying:** The system's behavior because of the bug is dehumanizing.*E.g.* Names are truncated or arbitrarily modified.
    4 **Disturbing:** It refuses to handle legitimate (authorized / legal) transactions. The ATM won't give you money. My credit card is declaredinvalid.
    5 **Serious:** It loses track of its transactions. Not just the transaction itself but the fact that the transaction occurred. Accountability islost.
    6 **Very Serious:** The bug causes the system to do the wrong transactions. Instead of losing your paycheck, the system credits it to another account or converts deposits towithdrawals.
    7 **Extreme:** The problems aren't limited to a few users or to few transaction types. They are frequent and arbitrary instead of sporadic infrequent) or for unusual cases.
    8 **Intolerable:** Long term unrecoverable corruption of the database occurs and the corruption is not easily discovered. Serious consideration is given to shutting the systemdown.
    9 **Catastrophic:** The decision to shut down is taken out of our hands because the systemfails.
    10 **Infectious:** What can be worse than a failed system? One that corrupt other systems even though it does not fall in itself ; that erodes the social physical environment; that melts nuclear reactors and startswar.

- Flexible severity rather than absolutes:

    - Quality can be measured as a combination of factors, of which number of bugs and their severity is only onecomponent.
    - Many organizations have designed and used satisfactory, quantitative, quality metrics.
    - Because bugs and their symptoms play a significant role in such metrics, as testing progresses, you see the quality rise to a reasonable value which is deemed to be safe to ship theproduct.
    - The factors involved in bug severityare:
        1. **Correction Cost:** Not so important because catastrophic bugs may be corrected easier and small bugs may take major time todebug.
        2. **Context and Application Dependency:** Severity depends on the context and the application in which it isused.
        3. **Creating Culture Dependency:** What's important depends on the creators of software and their cultural aspirations. Test tool vendors are more sensitive about bugs in their software then games softwarevendors.
        4. **User Culture Dependency:** Severity also depends on user culture. Naive users of PC software go crazy over bugs where as pros (experts) may just ignore.
        5. **The software development phase:** Severity depends on development phase. Any bugs gets more severe as it gets closer to field use and more severe the longer it has beenaround.

TAXONOMY OF BUGS:

- There is no universally correct way categorize bugs. The taxonomy is notrigid.
- A given bug can be put into one or another category depending on its history and the programmer's state ofmind.
- The major categories are: (1) Requirements, Features and Functionality Bugs (2) Structural Bugs (3) Data Bugs (4) Coding Bugs (5) Interface, Integration and System Bugs (6) Test and Test DesignBugs.

✓ **Requirements, Features and Functionality Bugs:** Various categories in Requirements, Features and Functionality bugsinclude:

1. Requirements and SpecificationsBugs:

- Requirements and specifications developed from them can be incomplete ambiguous,or self-contradictory. They can be misunderstood or impossible tounderstand.
- The specifications that don't have flaws in them may change while the design is in progress. The features are added, modified anddeleted.
- Requirements, especially, as expressed in specifications are a major source of expensive bugs.
- The range is from a few percentages to more than 50%, depending on the applicationand environment.
- What hurts most about the bugs is that they are the earliest to invade the system andthe last toleave.

2. Feature Bugs:

- Specification problems usually create corresponding featureproblems.
- A feature can be wrong, missing, or superfluous (serving no useful purpose). A missing feature or case is easier to detect and correct. A wrong feature could have deep design implications.
- Removing the features might complicate the software, consume more resources, and foster morebugs.

3. Feature InteractionBugs:

- Providing correct, clear, implementable and testable feature specifications is notenough.
- Features usually come in groups or related features. The features of each group and the interaction of features within the group are usually welltested.
- The problem is unpredictable interactions between feature groups or even between individual features. For example, your telephone is provided with call holding and call forwarding. The interactions between these two features may havebugs.
- Every application has its peculiar set of features and a much bigger set of unspecified feature interaction potentials and therefore result in feature interactionbugs.

Specification and Feature Bug Remedies:

- Most feature bugs are rooted in human to human communication problems. Onesolution is to use high-level, formal specification languages orsystems.
- Such languages and systems provide short term support but in the long run, does not solve theproblem.
- **Short term Support*:*** Specification languages facilitate formalization ofrequirements and inconsistency and ambiguityanalysis.
- **Long term Support*:*** Assume that we have a great specification language and that can be used to create unambiguous, complete specifications with unambiguous complete tests and consistent testcriteria.
- The specification problem has been shifted to a higher level but noteliminated.

Testing Techniques for functional bugs: Most functional test techniques- that is those techniques which are based on a behavioral description of software, such as transaction flow testing, syntax testing, domain testing, logic testing and state testing are useful in testing functional bugs.

✓ **Structural bugs:** Various categories in Structural bugsinclude:

1. Control and SequenceBugs:

- Control and sequence bugs include paths left out, unreachable code, improper nesting of loops, loop-back or loop termination criteria incorrect, missing process steps, duplicated processing, unnecessary processing, rampaging, GOTO's, ill-conceived (not properly planned) switches, spaghetti code, and worst of all, pachinkocode.
- One reason for control flow bugs is that this area is amenable (supportive) to theoretical treatment.
- Most of the control flow bugs are easily tested and caught in unittesting.
- Another reason for control flow bugs is that use of old code especially ALP & COBOL code are dominated by control flowbugs.

- Control and sequence bugs at all levels are caught by testing, especially structural testing, more specifically path testing combined with a bottom line functional test based on a specification.

2. Logic Bugs:

- Bugs in logic, especially those related to misunderstanding how case statements and logic operators behave singly andcombinations
- Also includes evaluation of boolean expressions in deeply nested IF-THEN-ELSE constructs.
- Ifthebugsarepartsoflogical(i.e.boolean)processingnotrelated to controlflow,they are characterized as processingbugs.
- If the bugs are parts of a logical expression (i.e. control-flow statement) which is used to direct the control flow, then they are categorized as control-flowbugs.

3. ProcessingBugs:

- Processingbugsincludearithmeticbugs,algebraic,mathematicalfunctionevaluation, algorithm selection and generalprocessing.
- ExamplesofProcessingbugsinclude:Incorrectconversionfromonedatarepresentation to other, ignoring overflow, improper use of greater-than-or-equaletc
- Although these bugs are frequent (12%), they tend to be caught in good unittesting.

4. InitializationBugs:

- Initialization bugs are common. Initialization bugs can be improper andsuperfluous.
- Superfluous bugs are generally less harmful but can affectperformance.
- Typical initialization bugs include: Forgetting to initialize the variables before first use, assuming that they are initialized elsewhere, initializing to the wrong format, representation or typeetc
- Explicit declaration of all variables, as in Pascal, can reduce some initializationproblems.

5. Data-Flow Bugs andAnomalies:

- Most initialization bugs are special case of data flowanomalies.
- A data flow anomaly occurs where there is a path along which we expect to do something unreasonable with data, such as using an uninitialized variable, attempting to use a variable before it exists, modifying and then not storing or using the result, or initializing twice without an intermediateuse.

✓ Databugs:

- Databugs includeallbugsthatarisefromthespecification of dataobjects,theirformats, the number of such objects, and their initialvalues.
- Data Bugs are at least as common as bugs in code, but they are often treated as if they did not exist atall.
- Code migrates data*:* Software is evolving towards programs in which more and more of the control and processing functions are stored intables.
- Because of this, there is an increasing awareness that bugs in code are only half thebattle and the data problems should be given equalattention.

Dynamic Data Vs Static data:

- Dynamic data are transitory. Whatever their purpose their lifetime is relatively short, typically the processing time of one transaction. A storage object may be used to hold dynamic data of different types, with different formats, attributes andresidues.
- Dynamic data bugs are due to leftover garbage in a shared resource. This can be handled in one of the three ways: (1) Clean up after the use by the user (2) Common Cleanup by the resource manager (3) No Cleanup
- Static Data are fixed in form and content. They appear in the source code or database directly or indirectly, for example a number, a string of characters, or a bitpattern.
- Compile time processing will solve the bugs caused by staticdata.

Information, parameter, and control:
Static or dynamic data can serve in one of three roles, or in combination of roles: as a parameter, for control, or for information.

Content, Structure and Attributes:

- Content can be an actual bit pattern, character string, or number put into a data structure. Content is a pure bit pattern and has no meaning unless it is interpreted by a hardware or software processor. All data bugs result in the corruption or misinterpretationofcontent.
- **Structure** relates to the size, shape and numbers that describe the data object, which is memory location used to store the content. (E.g. A twodimensionalarray).
- **Attributes** relates to the specification meaning that is the semantics associated with the contents of a data object. (E.g. an integer, an alphanumeric string, a subroutine). The severity and subtlety of bugs increases as we go from content to attributes because the things get less formal in thatdirection.

✓ Codingbugs:

- Codingerrorsofallkinds cancreateanyoftheotherkindofbugs.
- Syntax errors are generally not important in the scheme of things if the source language translator has adequate syntaxchecking.
- If a program has many syntax errors, then we should expect many logic and codingbugs.
- The documentation bugs are also considered as coding bugs which may misleadthe maintenance programmers.

✓ Interface, integration, and systembugs:
Various categories of bugs in Interface, Integration, and System Bugs are:

1. ExternalInterfaces:

- The external interfaces are the means used to communicate with theworld.
- Theseincludedevices,actuators,sensors,inputterminals,printers,andcommunication lines.
- The primary design criterion for an interface with outside world should berobustness.

- Allexternalinterfaces,humanor machineshouldemployaprotocol.Theprotocolmay be wrong or incorrectlyimplemented.
- Otherexternalinterfacebugsare:invalidtimingor sequenceassumptionsrelatedto externalsignals
- Misunderstanding external input or output formats.
- Insufficient tolerance to bad inputdata.

2. InternalInterfaces:

- Internalinterfacesareinprinciplenotdifferentfromexternalinterfacesbuttheyaremore controlled.
- A best example for internal interfaces is communicatingroutines.
- The external environment is fixed and the system must adapt to it but the internal environment,whichconsistsofinterfaceswithother components,canbenegotiated.
- Internal interfaces have the same problem as externalinterfaces.

3. HardwareArchitecture:

- Bugs related to hardware architecture originate mostly from misunderstanding how the hardware works.
- Examples of hardware architecture bugs: address generation error, i/o device operation / instruction error, waiting too long for a response, incorrect interrupt handlingetc.
- The remedy for hardware architecture and interface problems is twofold: (1) Good Programming and Testing (2) Centralization of hardware interface software in programs written by hardware interfacespecialists.

4. Operating SystemBugs:

- Program bugs related to the operating system are a combination of hardware architecture and interface bugs mostly caused by a misunderstanding of what it is the operating systemdoes.
- Use operating system interface specialists, and use explicit interface modules or macros for all operating systemcalls.
- This approach may not eliminate the bugs but at least will localize them and make testing easier.

5. Software Architecture:

- Software architecture bugs are the kind that called -interactive.
- Routines can pass unit and integration testing without revealing suchbugs.
- Many of them depend on load, and their symptoms emerge only when the system is stressed.
- Sample for such bugs: Assumption that there will be no interrupts, Failure to block or un block interrupts, Assumption that memory and registers were initialized or not initialized etc
- Careful integration of modules and subjecting the final system toa stress test are effective methods for thesebugs.

6. Control and Sequence Bugs (SystemsLevel):

These bugs include: Ignored timing, Assuming that events occur in a specified sequence, Working on data before all the data have arrived from disc, Waiting for an impossible combination of prerequisites, Missing, wrong, redundant or superfluous process steps.
The remedy for these bugs is highly structured sequencecontrol.
Specialize, internal, sequence control mechanisms arehelpful.

7. Resource ManagementProblems:

- Memory is subdivided into dynamically allocated resources such as buffer blocks, queue blocks, task control blocks, and overlaybuffers.
- External mass storage units such as discs, are subdivided into memory resourcepools.
- Some resource management and usage bugs: Required resource not obtained, Wrong resource used, Resource is already in use, Resource dead locketc
- **Resource Management Remedies:** A design remedy that prevents bugs is always preferable to a test method that discoversthem.
- The design remedy in resource management is to keep the resource structure simple: the fewest different kinds of resources, the fewest pools, and no private resource management.

8. IntegrationBugs:

- Integration bugs are bugs having to do with the integration of, and with the interfaces between, working and testedcomponents.
- These bugs results from inconsistencies or incompatibilities betweencomponents.
- The communication methods include data structures, call sequences, registers, semaphores, and communication links and protocols results in integrationbugs.
- The integration bugs do not constitute a big bug category (9%) they are expensive category because they are usually caught late in the game and because they force changes in several components and/or datastructures.

9. SystemBugs:

- System bugs covering all kinds of bugs that cannot be ascribed to a component or to their simple interactions, but result from the totality of interactions between many components such as programs, data, hardware, and the operatingsystems.
- There can be no meaningful system testing until there has been thorough component and integrationtesting.
- System bugs are infrequent (1.7%) but very important because they are often found only after the system has beenfielded.

✓ **TEST AND TEST DESIGNBUGS:**

- Testing: testers have no immunity to bugs. Tests require complicated scenarios and databases.
- They require code or the equivalent to execute and consequently they can havebugs.
- Test criteria: if the specification is correct, it is correctly interpreted and implemented, and a proper test has been designed; but the criterion by which the software's behavioris

judged may be incorrect or impossible. So, a proper test criteria has to be designed. The more complicated the criteria, the likelier they are to have bugs.

Remedies: The remedies of test bugs are:

1. Test Debugging: The first remedy for test bugs is testing and debugging the tests. Test debugging, when compared to program debugging, is easier because tests, when properly designed are simpler than programs and do not have to make concessions toefficiency.

2. Test Quality Assurance: Programmers have the right to ask how quality in independent testing ismonitored.

3. Test Execution Automation: The history of software bug removal and prevention is indistinguishable from the history of programming automation aids. Assemblers, loaders, compilers are developed to reduce the incidence of programming and operation errors. Test execution bugs are virtually eliminated by various test execution automationtools.

4. Test Design Automation: Just as much of software development has been automated, much test design can be and has been automated. For a given productivity rate, automation reducesthe

FLOW GRAPHS AND PATH TESTING

BASICS OF PATHTESTING:

- **PathTesting:**
  - Path Testing is the name given to a family of test techniques based onjudiciously selecting a set of test paths through theprogram.
  - If the set of paths are properly chosen then we have achieved some measure of test thoroughness. For example, pick enough paths to assure that every source statement has been executed at leastonce.
  - Path testing techniques are the oldest of all structural testtechniques.
  - Path testing is most applicable to new software for unit testing. It is a structural technique.
  - It requires complete knowledge of the program'sstructure.
  - It is most often used by programmers to unit test their owncode.
  - The effectiveness of path testing rapidly deteriorates as the size of thesoftware aggregate under testincreases.

- The BugAssumption:
  - The bug assumption for the path testing strategies is that somethinghasgone wrong with the software that makes it take a different path thanintended.
  - As an example "GOTO X" where "GOTO Y" had beenintended.
  - Structured programming languages prevent many of the bugs targeted by path testing: as a consequence the effectiveness for path testing for these languages is reduced and for old code in COBOL, ALP, FORTRAN and Basic, the pathtesting isindispensable.

- Control FlowGraphs:
  - The control flow graph is a graphical representation of a program'scontrol

structure. It uses the elements named process blocks, decisions, and junctions.

- The flow graph is similar to the earlier flowchart, with which it is not to be confused.

Flow Graph Elements: A flow graph contains four different types ofelements.

(1) Process Block (2) Decisions (3) Junctions (4) CaseStatements

1. ProcessBlock:
   - A process block is a sequence of program statementsuninterrupted by either decisions orjunctions.
   - It is a sequence of statements such that if any one of statementof the block is executed, then all statement thereof areexecuted.
   - Formally, a process block is a piece of straight line code of one statement or hundreds ofstatements.
   - A process has one entry and one exit. It can consists of a single statementorinstruction, asequenceofstatementsorinstructions,single entry/exit subroutine, a macro or function call, or a sequence of these.

2. Decisions:
   - A decision is a program point at which the control flowcan diverge.
   - Machinelanguageconditionalbranchandconditionalskip instructions are examples ofdecisions.
   - Most of the decisions are two-way but some are three way branches in controlflow.
3. Case Statements:
   - A case statement is a multi-way branch ordecisions.
   - Examples of case statement are a jump table in assembly language, and the PASCAL casestatement.
   - From the point of view of test design, there are no differences between Decisions and CaseStatements
4. Junctions:
   - A junction is a point in the program where the control flow can merge.
   - Examples of junctions are: the target of a jump or skip instruction in ALP, a label that is a target ofGOTO.

Figure 2.1: Flow graph Elements

Control Flow Graphs Vs Flowcharts:

- A program's flow chart resembles a control flowgraph.
- In flow graphs, we don't show the details of what is in a processblock.
- In flow charts every part of the process block isdrawn.
- The flowchart focuses on process steps, where as the flow graph focuses on control flow of theprogram.
- The act of drawing a control flow graph is a useful tool that can help us clarify the control flow and data flowissues.

Notational Evolution:

The control flow graph is simplified representation of the program's structure.The notation changes made in creation of control flow graphs:

- The process boxes weren't really needed. There is an implied process on every line joining junctions anddecisions.
- We don't need to know the specifics of the decisions, just the fact that there is abranch.
- The specific target label names aren't important-just the fact that they exist. So we can replace them by simple numbers.
- To understand this, we will go through an example (Figure 2.2) written in a FORTRAN like programming language called **Programming Design Language (PDL)**. The program's corresponding flowchart (Figure 2.3) and flowgraph (Figure 2.4) were also provided below for betterunderstanding.
- The first step in translating the program to a flowchart is shown in Figure 2.3, where we have the typical one-for-one classical flowchart. Note that complexity has increased, clarity has decreased, and that we had to add auxiliary labels (LOOP, XX, and YY), which have no actual program counterpart. In Figure 2.4 we merged the process steps and replaced them with the single processbox.

- We now have a control flow graph. But this representation is still too busy. We simplify the notation further to achieve Figure 2.5, where for the first time we can really see what the control flow lookslike.

CODE* (PDL)

```
        INPUT X, Y
        Z := X + Y
        V := X - Y
        IF Z >=Ø GOTO SAM
 JOE:   Z := Z - 1
 SAM:   Z := Z + V
        FOR U = Ø TO Z
        V(U),U(V) := (Z + V)*U
        IF V(U)= Ø GOTO JOE
        Z := Z - 1
        IF Z = Ø GOTO ELL
        U := U + 1
        NEXT U
        V(U-1):=V(U+1) + U(V-1)
ELL:    V(U+U(V)) := U + V
        IF U = V GOTO JOE
        IF U > V THEN U := Z
        Z := U
        END
```

* A contrived horror

Figure 2.2: Program Example (PDL)

Figure 2.3: One-to-one flowchart for example program in Figure 2.2

Figure 2.4: Control Flow graph for example in Figure 2.2

Figure 2.5: Simplified Flow graph Notation

Figure 2.6: Even Simplified Flow graph Notation

The final transformation is shown in Figure 2.6, where we've dropped the node numbers to achieve an even simpler representation. The way to work with control flow graphs is to use the simplest possible representation - that is, no more information than you need to correlate back to the source program or PDL.

LINKED LIST REPRESENTATION:

Although graphical representations of flow graphs are revealing, the details of the control flow inside a program they are often inconvenient.
In linked list representation, each node has a name and there is an entry on the list for each link in the flow graph. Only the information pertinent to the control flow is shown.

Linked List representation of Flow Graph:

```
1  (BEGIN)     : 3
2  (END)       :                Exit, no outlink
3  (Z>Ø?)      : 4 (FALSE)
               : 5 (TRUE)
4  (JOE)       : 5
5  (SAM)       : 6
6  (LOOP)      : 7
7  (V(U)=Ø?)   : 4 (TRUE)
               : 8 (FALSE)
8  (Z=Ø?)      : 9 (FALSE)
               :10 (TRUE)
9  (U=Z?)      : 6 (FALSE) = LOOP
               :10 (TRUE) = ELL
10 (ELL)       :11
11 (U=V?)      : 4 (TRUE) = JOE
               :12 (FALSE)
12 (U>V?)      :13 (TRUE)
               :13 (FALSE)
13 (END)       : 2 (END)
```

Figure 2.7: Linked List Control Flow graph Notation

FLOWGRAPH - PROGRAM CORRESPONDENCE:

A flow graph is a pictorial representation of a program and not the program itself, just as a topographic map.

You can't always associate the parts of a program in a unique way with flow graph parts because many program structures, such as if-then-else constructs, consists of a combination of decisions, junctions, and processes.

The translation from a flow graph element to a statement and vice versa is not always unique. (See Figure 2.8)

Figure 2.8: Alternative Flow graphs for same logic
(Statement "IF (A=0) AND (B=1) THEN . . .").

An improper translation from flow graph to code during coding can lead to bugs, and improper translation during the test design lead to missing test cases and causes undiscovered bugs.

FLOWGRAPH AND FLOWCHART GENERATION:

Flowcharts can be

1. Handwritten by theprogrammer.
2. Automaticallyproduced byaflowcharting programbasedonamechanicalanalysisof the sourcecode.
3. Semi automatically produced by a flow charting program based in part on structural analysis of the source code and in part on directions given by theprogrammer.

There are relatively few control flow graph generators.

PATH TESTING - PATHS, NODES AND LINKS:

Path: A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same junction, decision, or exit.

- Apathmaygothroughseveraljunctions,processes,ordecisions,oneormore times.
- Paths consist ofsegments.
- The segment is a link - a single process that lies between twonodes.
- A path segment is succession of consecutive links that belongs to somepath.
- The length of path measured by the number of links in it and not by the number of the instructions or statements executed along thatpath.
- The name of a path is the name of the nodes along thepath.

FUNDAMENTAL PATH SELECTION CRITERIA:

There are many paths between the entry and exit of a typical routine.
Every decision doubles the number of potential paths. And every loop multiplies the number of potential paths by the number of different iteration values possible for the loop.
Defining complete testing:

1. Exercise every path from entry toexit.
2. Exercise every statement or instruction at leastonce.
3. Exercise every branch and case statement, in each direction at leastonce.

If prescription 1 is followed then 2 and 3 are automatically followed. But it is impractical for most routines. It can be done for the routines that have no loops, in which it is equivalent to 2 and 3 prescriptions.

EXAMPLE*:* Here is the correct version.

For X negative, the output is X + A, while for X greater than or equal to zero, the output is X + 2A. Following prescription 2 and executing every statement, but not every branch, would not reveal the bug in the following incorrect version:

A negative value produces the correct answer. Every statement can be executed, but if the test cases do not force each branch to be taken, the bug can remain hidden. The next example uses a test based on executing each branch but does not force the execution of all statements:

The hidden loop around label 100 is not revealed by tests based on prescription 3 alone because no test forces the execution of statement 100 and the following GOTO statement. Furthermore, label 100 is not flagged by the compiler as an unreferenced label and the subsequent GOTO does not refer to an undefined label.

A **Static Analysis** (that is, an analysis based on examining the source code or structure) cannot determine whether a piece of code is or is not reachable. There could be subroutine calls with parameters that are subroutine labels, or in the above example there could be a GOTO that targeted label 100 but could never achieve a value that would send the program to that label.

Only a **Dynamic Analysis** (that is, an analysis based on the code's behavior while running - which is to say, to all intents and purposes, testing) can determine whether code is reachable or not and therefore distinguish between the ideal structure we think we have and the actual, buggy structure.

PATH TESTING CRITERIA:

Any testing strategy based on paths must at least both exercise every instruction and take branches in all directions.
A set of tests that does this is not complete in an absolute sense, but it is complete in the sense that anything less must leave somethinguntested.

So we have explored three different testing criteria or strategies out of a potentially infinite family of strategies.

i. Path Testing(P_{inf}):

1. Execute all possible control flow paths through the program: typically, this is restricted to all possible entry/exit paths through theprogram.
2. If we achieve this prescription, we are said to have achieved 100% path coverage. This is the strongest criterion in the path testing strategy family: it is generally impossible toachieve.

ii. Statement Testing(P1):

1. Execute all statements in the program at least once under some test. If we do enough tests to achieve this, we are said to have achieved 100% statementcoverage.
2. An alternate equivalent characterization is to say that we have achieved 100% node coverage. We denote this byC1.
3. This is the weakest criterion in the family: testing less than this for new software is unconscionable (unprincipled or cannot be accepted) and should becriminalized.

iii. Branch Testing(P2):

1. Execute enough tests to assure that every branch alternative has been exercised at least once under sometest.
2. If we do enough tests to achieve this prescription, then we have achieved 100%branch coverage.
3. An alternative characterization is to say that we have achieved 100% linkcoverage.
4. Forstructuredsoftware,branchtestingandthereforebranchcoveragestrictlyincludes statementcoverage.
5. We denote branch coverage byC2.

Commonsense and Strategies:

- Branch and statement coverage are accepted today as the minimum mandatory testing requirement.
- The question "why not use a judicious sampling of paths?, what is wrong with leaving some code, untested?" is ineffectual in the view of common sense and experience since: **(1.)** Not testing a piece of a code leaves a residue of bugs in the program in proportion to the size of the untested code and the probability of bugs. **(2.)** The high probability paths are alwaysthoroughlytestedifonlytodemonstrate thatthe systemworksproperly.
- **Which paths to be tested?** You must pick enough paths to achieve C1+C2. The question of what is the fewest number of such paths is interesting to the designer of test tools that help automate the path testing, but it is not crucial to the pragmatic (practical) design of tests. It is better to make many simple paths than a few complicatedpaths.

- Path SelectionExample:

Figure 2.9: An example flow graph to explain path selection

Practical Suggestions in Path Testing:

1. Draw the control flow graph on a single sheet ofpaper.
2. Make several copies - as many as you will need for coverage (C1+C2) and severalmore.
3. Use a yellow highlighting marker to trace paths. Copy the paths onto mastersheets.
4. Continuetracingpathsuntilall linesonthemastersheetarecovered,indicatingthatyou appear to have achievedC1+C2.
5. As you trace the paths, create a table that shows the paths, the coverage status ofeach process, and eachdecision.
6. The above paths lead to the following table considering Figure2.9:

PATHS	DECISIONS				PROCESS–LINK												
	4	6	7	9	a	b	c	d	e	f	g	h	i	j	k	l	m
abcde	YES	YES			✓	✓	✓	✓	✓								
abhkgde	NO	YES		NO	✓	✓		✓	✓		✓	✓			✓		
abhlibcde	NO,YES	YES		YES	✓	✓	✓	✓	✓			✓	✓			✓	
abcdfjgde	YES	NO,YES	YES		✓	✓	✓	✓	✓	✓	✓			✓			
abcdfmibcde	YES	NO,YES	NO		✓	✓	✓	✓	✓	✓			✓				✓

7. Afteryouhavetracedacoveringpathsetonthemastersheetandfilledinthetablefor every path, check thefollowing:

1. Does every decision have a YES and a NO in its column?(C2)
2. Has every case of all case statements been marked?(C2)
3. Is every three - way branch (less, equal, greater) covered?(C2)
4. Is every link (process) covered at least once?(C1)

8. Revised Path SelectionRules:

- Pick the simplest, functionally sensible entry/exitpath.
- Pick additional paths as small variation from previous paths. Pick paths that do not have loops rather than paths that do. Favor short paths that make sense over paths thatdon't.
- Pick additional paths that have no obvious functional meaning only if it's necessaryto provide coverage.
- Becomfortablewithyourchosenpaths.Playyourhunches(guesses)andgiveyour intuition free reign as long as you achieveC1+C2.
- Don't follow rules slavishly (blindly) - except forcoverage.

LOOPS:

Cases for a single loop: A Single loop can be covered with two cases: Looping and Not looping. But, experience shows that many loop-related bugs are not discovered by C1+C2. Bugs hide themselves in corners and congregate at boundaries - in the cases of loops, at or around the minimum or maximum number of times the loop can be iterated. The minimum number of iterations is often zero, but it need not be.

CASE 1: Single loop, Zero minimum, N maximum, No excluded values

1. Try bypassing the loop (zero iterations). If you can't, you either have a bug, or zero is not the minimum and you have the wrongcase.
2. Could the loop-control variable be negative? Could it appear to specify a negativenumber of iterations? What happens to such avalue?
3. One pass through theloop.
4. Two passes through theloop.
5. A typical number of iterations, unless covered by a previoustest.
6. One less than the maximum number ofiterations.
7. The maximum number ofiterations.
8. Attempt one more than the maximum number of iterations. What prevents the loop-control variable from having this value? What will happen with this value if it isforced?

CASE 2: Single loop, Non-zero minimum, No excluded values

1. Try one less than the expected minimum. What happens if the loop control variable'svalue is less than the minimum? What prevents the value from being less than theminimum?
2. The minimum number ofiterations.
3. One more than the minimum number ofiterations.
4. Once, unless covered by a previoustest.
5. Twice, unless covered by a previoustest.
6. A typicalvalue.
7. One less than the maximumvalue.
8. The maximum number ofiterations.
9. Attempt one more than the maximum number ofiterations.

CASE 3: Single loops with excluded values

- Treat single loops with excluded values as two sets of tests consisting of loops without excluded values, such as case 1 and 2above.
- Example, the total range of the loop control variable was 1 to 20, but that values 7, 8,9,10 were excluded. The two sets of tests are 1-6 and11-20.
- The test cases to attempt would be 0,1,2,4,6,7 for the first range and 10,11,15,19,20,21 for the secondrange.

Kinds of Loops: There are only three kinds of loops with respect to path testing:

- NestedLoops:
  The number of tests to be performed on nested loops will be the exponent of the tests performed on single loops.As we cannot always afford to test all combinations of nested loops' iterations values. Here's a tactic used to discard some of these values:
  1. Start at the inner most loop. Set all the outer loops to their minimumvalues.
  2. Test the minimum, minimum+1, typical, maximum-1 , and maximum for the innermost loop, while holding the outer loops at their minimum iteration parameter values. Expand the tests as required for out of range and excludedvalues.
  3. If you've done the outmost loop, GOTO step 5, else move out one loop and set it up as in step 2 with all other loops set to typicalvalues.
  4. Continue outward in this manner until all loops have beencovered.
  5. Do all the cases for all loops in the nestsimultaneously.

- ConcatenatedLoops:
  Concatenated loops fall between single and nested loops with respect to test cases. Two loops are concatenated if it's possible to reach one after exiting the other while still on a path from entrance to exit.
  If the loops cannot be on the same path, then they are not concatenated and can be treated as individual loops.

- Horrible Loops:
  A horrible loop is a combination of nested loops, the use of code that jumps into and out of loops, intersecting loops, hidden loops, and cross connected loops.
  Makes iteration value selection for test cases an awesome and ugly task, which is another reason such structures should be avoided.

Figure 2.10: Example of Loop types

Loop Testing Time:

Any kind of loop can lead to long testing time, especially if all the extreme value cases are to attempted (Max-1, Max, Max+1).

- This situation is obviously worse for nested and dependent concatenatedloops.
- Consider nested loops in which testing the combination of extreme values lead to long test times. Several options to dealwith:
- Prove that the combined extreme cases are hypothetically possible, they are not possible in the realworld

- Put in limits or checks that prevent the combined extreme cases. Then you have to test the software that implements such safetymeasures.

PREDICATES, PATH PREDICATES AND ACHIEVABLE PATHS:

PREDICATE: The logical function evaluated at a decision is called Predicate. The direction taken at a decision depends on the value of decision variable. Some examples are: A>0, x+y>=90.....

PATH PREDICATE: A predicate associated with a path is called a Path Predicate. For example, "x is greater than zero", "x+y>=90", "w is either negative or equal to 10 is true" is a sequence of predicates whose truth values will cause the routine to take a specificpath.

MULTIWAY BRANCHES:

- The path taken through a multiway branch such as a computed GOTO's, case statement, or jump tables cannot be directly expressed in TRUE/FALSEterms.
- Although, it is possible to describe such alternatives by using multi valued logic, an expedient (practical approach) is to express multiway branches as an equivalent set of if..then..else statements.
- For example a three way case statement can be written as: If case=1 DO A1 ELSE (IF Case=2 DO A2 ELSE DO A3ENDIF)ENDIF.

INPUTS:

- In testing, the word input is not restricted to direct inputs, such as variables in a subroutine call, but includes all data objects referenced by the routine whose values are fixed prior to enteringit.
- For example, inputs in a calling sequence, objects in a data structure, values left in registers, or any combination of objecttypes.
- The input for a particular test is mapped as a one dimensional array called as an Input Vector.

PREDICATE INTERPRETATION:

- The simplest predicate depends only on inputvariables.
- For example if x1,x2 are inputs, the predicate might be x1+x2>=7, given the values of x1 and x2 the direction taken through the decision is based on the predicate is determined at input time and does not depend onprocessing.
- Another example, assume a predicate x1+y>=0 that along a path prior to reaching this predicate we had the assignment statement y=x2+7. although our predicate depends on processing, we can substitute the symbolic expression for y to obtain an equivalent predicate x1+x2+7>=0.
- The act of symbolic substitution of operations along the path in order to express the predicate solely in terms of the input vector is called **predicateinterpretation.**
- Sometimes the interpretation may depend on the path; forexample,

INPUTX
ON X GOTO A, B, C, ...

A: Z := 7 @ GOTO HEM
B: Z := -7 @ GOTO HEM
C: Z := 0 @ GOTO HEM
.....
HEM: DO SOMETHING
.....
HEN: IF Y + Z > 0 GOTO ELL ELSE GOTO EMM

The predicate interpretation at HEN depends on the path we took through the first multiway branch. It yields for the three cases respectively, if Y+7>0, Y-7>0, Y>0.

- The path predicates are the specific form of the predicates of the decisions along theselected path afterinterpretation.

INDEPENDENCE OF VARIABLES AND PREDICATES:

- The path predicates take on truth values based on the values of input variables, either directly orindirectly.
- If a variable's value does not change as a result of processing, that variable is independent of theprocessing.
- If the variable's value can change as a result of the processing, the variable is process dependent.
- A predicate whose truth value can change as a result of the processing is said to be **process dependent** and one whose truth value does not change as a result of the processing is **processindependent.**
- Process dependence of a predicate does not always follow from dependence of the input variables on which that predicate isbased.

CORRELATION OF VARIABLES AND PREDICATES:

Two variables are correlated if every combination of their values cannot be independently specified.

Variables whose values can be specified independently without restriction are called uncorrelated.

A pair of predicates whose outcomes depend on one or more variables in common are said to be correlated predicates.

For example, the predicate X==Y is followed by another predicate X+Y == 8. If we select X and Y values to satisfy the first predicate, we might have forced the 2nd predicate's truth value to change.

- Every path through a routine is achievable only if all the predicates in that routine are uncorrelated.

PATHPREDICATEEXPRESSIONS:

- A path predicate expression is a set of boolean expressions, all of which must be satisfied to achieve the selectedpath.
- Example:

  X1+3X2+17>=0
  X3=17
  X4-X1>=14X2

- Anysetofinputvaluesthatsatisfyalloftheconditionsofthepathpredicateexpressionwill force the routine to thepath.
- Sometimes a predicate can have an OR init.
- Example:

A: X5 > 0	E: X6 < 0
B: X1 + 3X2 + 17 >= 0	B: X1 + 3X2 + 17 >= 0
C: X3 = 17	C: X3 = 17
D: X4 - X1 >= 14X2	D: X4 - X1 >= 14X2

- Boolean algebra notation to denote the booleanexpression:
  ABCD+EBCD=(A+E)BCD

PREDICATE COVERAGE:

- **Compound Predicate:** Predicates of the form A OR B, A AND B and more complicated Boolean expressions are called as compoundpredicates.
- Sometimes even a simple predicate becomes compound after interpretation. Example: the predicate if (x=17) whose opposite branch is if x.NE.17 which is equivalent to x>17. Or. X<17.
- Predicate coverage is being the achieving of all possible combinations of truth values corresponding to the selected path have been explored under sometest.
- As achieving the desired direction at a given decision could still hide bugs in the associated predicates

TESTING BLINDNESS:

- Testing Blindness is a pathological (harmful) situation in which the desired path is achieved for the wrongreason.
- There are three types of TestingBlindness:

- AssignmentBlindness:
  - Assignment blindness occurs when the buggy predicate appears to work correctly because the specific value chosen for an assignment statement works with both the correct and incorrectpredicate.
  - ForExample:

Correct	**Buggy**
X = 7 ..... if Y > 0 then ...	X = 7 ..... if X+Y > 0 then ...

  - If the test case sets Y=1 the desired path is taken in either case, but there is still abug.

- EqualityBlindness:
  - Equality blindness occurs when the path selected by a prior predicate results in a value that works both for the correct and buggypredicate.

- ForExample:

Correct	Buggy
if Y = 2 then ..... if X+Y > 3 then ...	if Y = 2 then ..... if X > 1 then ...

- The first predicate if y=2 forces the rest of the path, so that for any positive value of x. the path taken at the second predicate will be the same for the correct and buggyversion.

 SelfBlindness:

- Self blindness occurs when the buggy predicate is a multiple of the correct predicate and as a result is indistinguishable along thatpath.
- ForExample:

Correct	Buggy
X = A ..... if X-1 > 0 then ...	X = A ..... if X+A-2 > 0 then ...

1. The assignment (x=a) makes the predicates multiples of each other, so the direction taken is the same for the correct and buggy version.

 PATHSENSITIZING:

- **Review: achievable and unachievablepaths:**

1. We want to select and test enough paths to achieve a satisfactory notion of testcompleteness such asC1+C2.
2. Extract the programs control flow graph and select a set of tentative coveringpaths.
3. For any path in that set, interpret the predicates along the path as needed to express them in terms of the input vector. In general individual predicates are compound or may become compound as a result ofinterpretation.
4. Trace the path through, multiplying the individual compound predicates to achieve a boolean expression suchas

(A+BC) (D+E) (FGH) (IJ) (K) (l) (L).

5. Multiply out the expression to achieve a sum of productsform:

ADFGHIJKL+AEFGHIJKL+BCDFGHIJKL+BCEFGHIJKL

6. Each product term denotes a set of inequalities that if solved will yield an input vector that will drive the routine along the designatedpath.
7. Solveanyone oftheinequalitysetsfor thechosenpathandyouhavefoundasetofinput values for thepath.
8. If you can find a solution, then the path isachievable.
9. If you can't find a solution to any of the sets of inequalities, the path is unachievable.
10. The act of finding a set of solutions to the path predicate expression is called **PATH SENSITIZATION**.

o HEURISTIC PROCEDURES FOR SENSITIZINGPATHS:

1. This is a workable approach, instead of selecting the paths without considering how to sensitize, attempt to choose a covering path set that is easy to sensitize and pick hard to sensitize paths only as you must to achievecoverage.
2. Identify all variables that affect thedecision.
3. Classify the predicates as dependent orindependent.
4. Start the path selection with un correlated, independentpredicates.
5. If coverage has not been achieved using independent uncorrelated predicates, extendthe path set using correlatedpredicates.
6. If coverage has not been achieved extend the cases to those that involvedependent predicates.
7. Last, use correlated, dependentpredicates.

 PATHINSTRUMENTATION:

1. Path instrumentation is what we have to do to confirm that the outcome was achieved bythe intendedpath.
2. **Co-incidentalCorrectness:**Thecoincidentalcorrectnessstandsforachievingthedesired outcome for wrongreason.

Figure 2.11: Coincidental Correctness

The above figure is an example of a routine that, for the (unfortunately) chosen input value (X = 16), yields the same outcome (Y = 2) no matter which case we select. Therefore, the tests chosen this way will not tell us whether we have achieved coverage. For example, the five cases could be totally jumbled and still the outcome would be the same. **Path Instrumentation** is what we have to do to confirm that the outcome was achieved by the intendedpath.

 The types of instrumentation methods include:

1. Interpretive TraceProgram:

- An interpretive trace program is one that executes every statement in order and records the intermediate values of all calculations, the statement labels traversedetc.
- If we run the tested routine under a trace, then we have all the information we need to confirm the outcome and, furthermore, to confirm that it was achieved by the intended path.
- The trouble with traces is that they give us far more information than we need. In fact,the typical trace program provides so much information that confirming the path from its massive output dump is more work than simulating the computer by hand to confirm the path.

2. Traversal Marker or LinkMarker:

- A simple and effective form of instrumentation is called a traversal marker or linkmarker.
- Name every link by a lower caseletter.
- Instrument the links so that the link's name is recorded when the link isexecuted.
- The succession of letters produced in going from the routine's entry to its exit should, if there are no bugs, exactly correspond to the pathname.

Figure 2.12: Single Link Marker Instrumentation

- **Why Single Link Markers aren't enough:** Unfortunately, a single link marker may not do the trick because links can be chewed by openbugs.

Figure 2.13: Why Single Link Markers aren't enough.

We intended to traverse the ikm path, but because of a rampaging GOTO in the middle of the m link, we go to process B. If coincidental correctness is against us, the outcomes will be the same and we won't know about the bug.

- Two Link MarkerMethod:
  The solution to the problem of single link marker method is to implement two markers per link: one at the beginning of each link and on at the end.
  The two link markers now specify the path name and confirm both the beginning and end of the link.

Figure 2.14: Double Link Marker Instrumentation

- **Link Counter:** A less disruptive (and less informative) instrumentation method isbased on counters. Instead of a unique link name to be pushed into a string when the link is traversed, we simply increment a link counter. We now confirm that the path length is as expected. The same problem that led us to double link markers also leads us to double link counters.

5. bug count - be it for software or be it fortests.

PART A(2 MARKS)

1. Defineprocess.
   Process, in the software engineering domain, is the set of methods, practices, standards, documents, activities, policies, and procedures that software engineers use to develop and maintain a software system and its associated artifacts, such as project and test plans, design documents, code and manuals.

2. DefineValidation.
   Validation is the process of evaluating a software system or component during, or at the end of, the development cycle in order to determine whether it satisfies specified requirements.

3. DefineVerification.
   Verification is the process of evaluating a software system or component to determine whether the product of a given development phase satisfy the conditions imposed at the start of that phase.

4. DefineTesting.
   Testing is generally described as a group of procedures carried out to evaluate some aspects of a piece of software.
   (OR)
   Testing can be described as a process used for revealing defects in software, and for establishing that the software has attained a specified degree of quality with respect to selected attributes.

5. DefineDebugging.
   Debugging, or fault localization is the process of 9 Locating the fault or defect. 9 Repairing the codes. 9 Retesting the code.

6. List out the levels of the testing maturitymodel..

   Level 1: Initial.
   Level 2: Phase Definition.
   Level 3: Integration.
   Level 4: Management and Measurement.
   Level 5: Optimization/Defect prevention and quality control.

7. Define Errors.
   An error is a mistake, misconception, or misunderstanding on the part of a software developer.

8. DefineFaults.
   A fault (defect) is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification.

9. DefineFailures.

A failure is the inability of a software system or component to perform its required functions within specified performance requirements.

10. Define Testcase.

A test case in the practical sense is a test- related item which contain the following information: A set of test inputs.
Execution
conditions.
Expected outputs.

11. DefineTest.

A test is a group of related test cases, or a group of related test cases and test procedures.

12. Define TestOracle.

A test oracle is a document, or piece of software that allows tester to determine whether a test has been passed or failed.

13. DefineQuality.

Two concise definitions for quality
. Quality relates to the degree to which a system, system component, or process meets specified requirements.
Quality relates to the degree to which a system, system component, or process meets customer or user needs, or expectations.

14. DefineMetric.

A metric is a quantitative measure of the degree to which a system, system component, or process possesses a given attribute.

15. Define QualityMetric.

A quality metric is a quantitative measure of the degree to which an item possesses a quality attribute

16. DefineReview.

A review is a group meeting whose purpose is to evaluate artifact or a set of software artifacts.

17. DefinePrecondition.

A precondition is a condition that must be true in order for a software component to operate properly. Eg; number_of_coins > =0 20. Define Post condition. A post condition is a condition that must be true when a software component completes its operation properly. Eg; number_of_dollars, number_of_cents >
= 0

PART B(10 MARKS)

1. Differentiate between control flow graph andflowchart.
2. Demonstrate various kinds of loops with respect to path testing using neatdiagrams
3. Explain briefly about structural bugs and codingbug.
4. List the goals oftesting

5. Define testing blindness and list itscategories

UNIT II

TRANSACTION FLOW TESTING AND DATA FLOW TESTING

INTRODUCTION:

- A transaction is a unit of work seen from a system user's point ofview.
- A transaction consists of a sequence of operations, some of which areperformed by a system, persons or devices that are outside of thesystem.
- Transaction begins with Birth-that is they are created as a result of some external act.
- At the conclusion of the transaction's processing, the transaction is no longer in thesystem.
- **Example of a transaction:** A transaction for an online informationretrieval system might consist of the following steps ortasks:
    - Accept input (tentativebirth)
    - Validate input(birth)
    - Transmit acknowledgement torequester
    - Do inputprocessing
    - Searchfile
    - Request directions fromuser
    - Acceptinput
    - Validateinput
    - Processrequest
    - Updatefile
    - Transmitoutput
    - Record transaction in log and clean up(death)

- TRANSACTION FLOWGRAPHS:

    - Transaction flows are introduced as a representation of a system'sprocessing.
    - The methods that were applied to control flow graphs are then used for functional testing.
    - Transaction flows and transaction flow testing are to the independent system tester what control flows are path testing are to theprogrammer.
    - The transaction flow graph is to create a behavioral model of the program that leads to functionaltesting.
    - Thetransaction flowgraphisamodelofthestructureofthesystem'sbehavior (functionality).
    - An example of a Transaction Flow is asfollows:

Figure 3.1: An Example of a Transaction Flow

- **USAGE:**
  - Transactionflowsareindispensableforspecifying requirementsofcomplicated systems, especially onlinesystems.
  - A big system such as an air traffic control or airline reservation system, has not hundreds, but thousands of different transactionflows.
  - The flows are represented by relatively simple flowgraphs, many of which havea single straight-throughpath.
  - Loops are infrequent compared to controlflowgraphs.
  - The most common loop is used to request a retry after user input errors. An ATM system, for example, allows the user to try, say three times, and will take the card away the fourthtime.

- COMPLICATIONS:
  - Insimplecases, thetransactionshaveauniqueidentityfromthetimethey're created to the time they'recompleted.
  - In many systems the transactions can give birth to others, and transactionscan alsomerge.
  - **Births:** There are three different possible interpretations of the decisionsymbol, or nodes with two or more out links. It can be a Decision, Biosis or aMitosis.
    1. **Decision:** Here the transaction will take one alternative or the other alternative but not both. (See Figure 3.2(a))
    2. **Biosis:** Here the incoming transaction gives birth to a new transaction, and both transaction continue on their separate paths, and the parent retains it identity. (See Figure 3.2(b))
    3. **Mitosis:** Here the parent transaction is destroyed and two new transactions are created.(See Figure 3.2(c))

Figure 3.2: Nodes with multiple outlinks

Mergers: Transaction flow junction points are potentially as troublesome as transaction flow splits. There are three types of junctions: (1) Ordinary Junction (2) Absorption (3) Conjugation

1 **Ordinary Junction:** An ordinary junction which is similar to the junction in a control flow graph. A transaction can arrive either on one link or the other. (See Figure 3.3(a))
2 **Absorption:** In absorption case, the predator transaction absorbs prey transaction. The prey gone but the predator retains its identity. (See Figure 3.3(b))
3 **Conjugation:** In conjugation case, the two parent transactions merge to form a new daughter. In keeping with the biological flavor this case is called as conjugation.(See Figure 3.3(c))

Figure 3.3: Transaction Flow Junctions and Mergers

We have no problem with ordinary decisions and junctions. Births, absorptions, and conjugations are as problematic for the software designer as they are for the software modeler and the test designer; as a consequence, such points have more than their share of bugs. The common problems are: lost daughters, wrongful deaths, and illegitimate births.

TRANSACTION FLOW TESTING TECHNIQUES:

- **GET THE TRANSACTIONSFLOWS:**
  - Complicated systems that process a lot of different, complicated transactions should haveexplicitrepresentationsofthetransactionsflows, ortheequivalent.
  - Transaction flows are like control flow graphs, and consequently we should expect to have them in increasing levels ofdetail.
  - Thesystem'sdesigndocumentationshouldcontainanoverviewsectionthat details the main transactionflows.
  - Detailed transaction flows are a mandatory pre requisite to the rational design of a system's functionaltest.

- INSPECTIONS, REVIEWS ANDWALKTHROUGHS:
  - Transaction flows are natural agenda for system reviews orinspections.
  - In conducting the walkthroughs, youshould:

        - Discuss enough transaction types to account for 98%-99% ofthe transaction the system is expected toprocess.
        - Discuss paths through flows in functional rather than technicalterms.
        - Ask the designers to relate every flow to the specification and to show howthattransaction,directlyorindirectly,followsfromtherequirements.
    - Make transaction flow testing the corner stone of system functional testing just as path testing is the corner stone of unittesting.
    - Select additional flow paths for loops, extreme values, and domainboundaries.
    - Design more test cases to validate all births anddeaths.
    - Publishand distributetheselectedtestpathsthroughthetransaction flowsasearly as possible so that they will exert the maximum beneficial effect ontheproject.

- PATHSELECTION:
    - Select a set of covering paths (c1+c2) using the analogous criteria you usedfor structural pathtesting.
    - Selectacoveringsetofpathsbasedonfunctionallysensibletransactionsasyou would for control flowgraphs.
    - Try to find the most tortuous, longest, strangest path from the entry to the exit of the transactionflow.

- PATHSENSITIZATION:
    - Most of the normal paths are very easy to sensitize-80% - 95% transaction flow coverage (c1+c2) is usually easy toachieve.
    - The remaining small percentage is often verydifficult.
    - Sensitization is the act of defining the transaction. If there are sensitization problems on the easy paths, then bet on either a bug in transaction flows or a designbug.

- PATHINSTRUMENTATION:
    - Instrumentation plays a bigger role in transaction flow testing than in unit path testing.
    - The information of the path taken for a given transaction must be kept with that transaction and can be recorded by a central transaction dispatcher or by the individual processingmodules.
    - In some systems, such traces are provided by the operating systems or a running log.

BASICS OF DATA FLOW TESTING:

- **DATA FLOWTESTING:**
    - Data flow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status ofdataobjects.
    - For example, pick enough paths to assure that every data object has been initialized prior to use or that all defined objects have been used forsomething.
    - **Motivation:** It is our belief that, just as one would not feel confident about a program without executing every statement in it as part of some test, oneshould

not feel confident about a program without having seen the effect of using the value produced by each and every computation.

- DATA FLOWMACHINES:
  - There are two types of data flow machines with different architectures. (1) Von Neumann machines (2) Multi-instruction, multi-data machines(MIMD).
  - Von Neumann Machine Architecture:
    - Most computers today are von-neumannmachines.
    - This architecture features interchangeable storage of instructions anddata in the same memoryunits.
    - The Von Neumann machine Architecture executes one instruction ata time in the following, micro instructionsequence:
      - Fetch instruction from memory
      - Interpretinstruction
      - Fetchoperands
      - Process orExecute
      - Storeresult
      - Increment programcounter
      - GOTO1
  - Multi-instruction, Multi-data machines (MIMD)Architecture:
    - These machines can fetch several instructions and objects inparallel.
    - They can also do arithmetic and logical operations simultaneouslyon different dataobjects.
    - The decision of how to sequence them depends on thecompiler.

- BUGASSUMPTION:

  The bug assumption for data-flow testing strategies is that control flow is generally correct and that something has gone wrong with the software so that data objects are not available when they should be, or silly things are being done to dataobjects.
  - Also, if there is a control-flow problem, we expect it to have symptoms that can be detected by data-flowanalysis.
  - Although we'll be doing data-flow testing, we won't be using data flow graphs as such. Rather, we'll use an ordinary control flow graph annotated to show what happens to the data objects of interest at themoment.

- DATA FLOWGRAPHS:
  - The data flow graph is a graph consisting of nodes and directedlinks.
  - We will use a control graph to show what happens to data objects of interest at thatmoment.
  - Our objective is to expose deviations between the data flows we have and thedata flows wewant.

Figure 3.4: Example of a data flow graph

- **Data Object State andUsage:**
  - Data Objects can be created, killed andused.
  - They can be used in two distinct ways: (1) In a Calculation (2) As a part of a Control FlowPredicate.
  - The following symbols denote thesepossibilities:
    1. **Defined:** d - defined, created, initializedetc
    2. **Killed or undefined:** k - killed, undefined, releasedetc
    3. **Usage:** u - used for something (c - used in Calculations, p - used in a predicate)
  - 1. Defined(d):
    - An object is defined explicitly when it appears in a data declaration.
    - Or implicitly when it appears on the left hand side of the assignment.
    - It is also to be used to mean that a file has beenopened.
    - A dynamically allocated object has beenallocated.
    - Something is pushed on to thestack.
    - A recordwritten.

    2. Killed or Undefined(k):
    - An object is killed on undefined when it is released or otherwise made unavailable.

- When its contents are no longer known with certitude (with absolute certainty /perfectness).
- Release of dynamically allocated objects back to the availability pool.
- Return ofrecords.
- The old top of the stack after it ispopped.
- An assignment statement can kill and redefine immediately. For example, if A had been previously defined and we do a new assignment such as A : = 17, we have killed A's previous value and redefinedA

3. Usage(u):

- A variable is used for computation (c) when it appears on theright hand side of an assignmentstatement.
- A file record is read orwritten.
- It is used in a Predicate (p) when it appears directly in apredicate.

DATA FLOW ANOMALIES:

An anomaly is denoted by a two-character sequence of actions. For example, ku means that the object is killed and then used, where as dd means that the object is defined twice without an intervening usage.

What is an anomaly is depend on the application.

There are nine possible two-letter combinations for d, k and u. some are bugs, some are suspicious, and some are okay.

1. **dd** :- probably harmless but suspicious. Why define the object twice without anintervening usage?
2. **dk** :- probably a bug. Why define the object without usingit?
3. **du** :- the normal case. The object is defined and thenused.
4. **kd** :- normal situation. An object is killed and thenredefined.
5. **kk** :- harmless but probably buggy. Did you want to be sure it was reallykilled?
6. **ku** :- a bug. the object doesnotexist.
7. **ud** :- usually not a bug because the language permits reassignment at almost anytime.
8. **uk** :-normalsituation.
9. **uu** :-normalsituation.

In addition to the two letter situations, there are six single letter situations.We will use a leading dash to mean that nothing of interest (d,k,u) occurs prior to the action noted along the entry-exit path of interest.

A trailing dash to mean that nothing happens after the point of interest to the exit.

They possible anomalies are:

1. **-k** :- possibly anomalous because from the entrance to this point on the path,the variable had not been defined. We are killing a variable that does notexist.
2. **-d** :- okay. This is just the first definition along thispath.
3. **-u** :- possibly anomalous. Not anomalous if the variable is global and hasbeen previouslydefined.

4 **k-** :- not anomalous. The last thing done on this path was to kill thevariable.

5 **d-** :- possibly anomalous. The variable was defined and not used on this path. But this could be a globaldefinition.

6 **u-** :- not anomalous. The variable was used but not killed on this path. Although this sequence is not anomalous, it signals a frequent kind of bug. If d and k mean dynamic storage allocation and return respectively, this could be an instance in which a dynamically allocated object was not returned to the pool afteruse.

DATA FLOW ANOMALY STATE GRAPH:

Data flow anomaly model prescribes that an object can be in one of four distinct states:

0. **K** :- undefined, previously killed, doesnotexist
1. **D** :- defined but not yet used foranything
2. **U** :- has been used for computation or inpredicate
3. **A** :-anomalous

These capital letters (K, D, U, A) denote the state of the variable and should not be confused with the program action, denoted by lower case letters.

Unforgiving Data - Flow Anomaly Flow Graph: Unforgiving model, in which once a variable becomes anomalous it can never return to a state of grace.

Figure 3.5: Unforgiving Data Flow Anomaly State Graph

Assume that the variable starts in the K state - that is, it has not been defined or does not exist. If an attempt is made to use it or to kill it (e.g., say that we're talking about opening, closing, and using files and that 'killing' means closing), the object's state becomes anomalous (state A) and, once it is anomalous, no action can return the variable to a working state.

If it is defined (d), it goes into the D, or defined but not yet used, state. If it has been defined (D) and redefined (d) or killed without use (k), it becomes anomalous, while usage (u) brings it to the U state. If in U, redefinition (d) brings it to D, u keeps it in U, and k kills it.

Forgiving Data - Flow Anomaly Flow Graph: Forgiving model is an alternate model where redemption (recover) from the anomalous state is possible

Figure 3.6: Forgiving Data Flow Anomaly State Graph

This graph has three normal and three anomalous states and he considers the kk sequence not to be anomalous. The difference between this state graph and Figure 3.5 is that redemption is possible. A proper action from any of the three anomalous states returns the variable to a useful working state.

The point of showing you this alternative anomaly state graph is to demonstrate that the specifics of an anomaly depends on such things as language, application, context, or even your frame of mind. In principle, you must create a new definition of data flow anomaly (e.g., a new state graph) in each situation. You must at least verify that the anomaly definition behind the theory or imbedded in a data flow anomaly test tool is appropriate to your situation.

STATIC Vs DYNAMIC ANOMALY DETECTION:

Static analysis is analysis done on source code without actually executing it. For example: source code syntax error detection is the static analysis result.

Dynamic analysis is done on the fly as the program is being executed and is based on intermediate values that result from the program's execution. For example: a division by zero warning is the dynamicresult.

If a problem, such as a data flow anomaly, can be detected by static analysis methods, then it doesn’t belongs in testing - it belongs in the language processor.
There is actually a lot more static analysis for data flow analysis for data flow anomalies going on in current languageprocessors.

For example, language processors which force variable declarations can detect (-u) and (ku) anomalies.But still there are many things for which current notions of static analysis are INADEQUATE.

Why Static Analysis isn't enough? There are many things for which current notions of static analysis are inadequate. They are:

- **Dead Variables:** Although it is often possible to prove that a variable is dead or alive at a given point in the program, the general problem isunsolvable.

- **Arrays:** Arrays are problematic in that the array is defined or killed as a single object, but reference is to specific locations within the array. Array pointers are usually dynamically calculated, so there's no way to do a static analysis to validate the pointer value. In many languages, dynamically allocated arrays contain garbage unless explicitly initialized and therefore, -u anomalies arepossible.

- **Records and Pointers:** The array problem and the difficulty with pointers is a special case of multipart data structures. We have the same problem with records and the pointers to them. Also, in many applications we create files and their names dynamically and there's no way to determine, without execution, whether such objects are in the proper state on a given path or, for that matter, whether they exist atall.

- **Dynamic Subroutine and Function Names in a Call:** subroutine or function name is a dynamic variable in a call. What is passed, or a combination of subroutine names and data objects, is constructed on a specific path. There's no way, without executing the path, to determine whether the call is correct ornot.

- **False Anomalies:** Anomalies are specific to paths. Even a "clear bug" such as ku may not be a bug if the path along which the anomaly exist is unachievable. Such "anomalies" are false anomalies. Unfortunately, the problem of determining whether a path is or is not achievable isunsolvable.

- **Recoverable Anomalies and Alternate State Graphs:** What constitutes an anomaly depends on context, application, and semantics. How does the compiler know which model I have in mind? It can't because the definition of "anomaly" is not fundamental. The language processor must have a built-in anomaly definition with which you may or maynot (withgood reason)agree.

- **Concurrency, Interrupts, System Issues:** As soon as we get away from the simple single-task uniprocessor environment and start thinking in terms of systems, most anomaly issues become vastly morecomplicated.

  How often do we define or create data objects at an interrupt level so that they can be processed by a lower-priority routine? Interrupts can make the "correct" anomalous and the "anomalous" correct. True concurrency (as in an MIMD machine) and pseudo concurrency (as in multiprocessing) systems can do the same to us. Much of integration and system testing is aimed at detecting data-flow anomalies that cannot be detected in the context of a single routine.

  Although static analysis methods have limits, they are worth using and a continuing trend in language processor design has been better static analysis methods, especially for data flow anomaly detection. That's good because it means there's less for us to do as testers and we have far too much to do as it is.

DATA FLOW MODEL:

The data flow model is based on the program's control flow graph - Don't confuse that with the program's data flow graph.
Here we annotate each link with symbols (for example, d, k, u, c, and p) or sequences of symbols (for example, dd, du, ddd) that denote the sequence of data operations on that link with respect to the variable of interest. Such annotations are called link weights.
The control flow graph structure is same for every variable: it is the weights that change.

Components of the model:

1. To every statement there is a node, whose name is unique. Every node has at least one outlink and at least one inlink except for exit nodes and entrynodes.
2. Exit nodes are dummy nodes placed at the outgoing arrowheads of exit statements (e.g., END, RETURN), to complete the graph. Similarly, entry nodes are dummy nodes placed at entry statements (e.g., BEGIN) for the samereason.
3. The outlink of simple statements (statements with only one outlink) are weighted by the proper sequence of data-flow actions for that statement. Note that the sequence can consist of more than one letter. For example, the assignment statement A:= A + B in most languages is weighted by cd or possibly ckd for variable A. Languages that permit multiple simultaneous assignments and/or compound statements can have anomalies within the statement. The sequence must correspond to the order in which the object code will be executed for thatvariable.
4. Predicate nodes (e.g., IF-THEN-ELSE, DO WHILE, CASE) are weighted with the p - use(s) on every outlink, appropriate to thatoutlink.
5. Every sequence of simple statements (e.g., a sequence of nodes with one inlink and one outlink) can be replaced by a pair of nodes that has, as weights on the link between them, the concatenation of linkweights.
6. If there are several data-flow actions on a given link for a given variable, then the weight of the link is denoted by the sequence of actions on that link for thatvariable.
7. Conversely, a link with several data-flow actions on it can be replaced by a succession of equivalent links, each of which has at most one data-flow action for anyvariable.

Let us consider the example:

CODE* (PDL)

```
        INPUT X, Y
        Z := X + Y
        V := X − Y
        IF Z >=Ø GOTO SAM
JOE:    Z := Z − 1
SAM:    Z := Z + V
        FOR U = Ø TO Z
        V(U),U(V) := (Z + V)*U
        IF V(U)= Ø GOTO JOE
        Z := Z − 1
        IF Z = Ø GOTO ELL
        U := U + 1
        NEXT U
        V(U−1):=V(U+1) + U(V−1)
ELL:V(U+U(V)) := U + V
        IF U = V GOTO JOE
        IF U > V THEN U := Z
        Z := U
        END
```

* A contrived horror

Figure 3.7: Program Example (PDL)

Figure 3.8: Unannotated flow graph for example program in Figure 3.7

Figure 3.9: Control flow graph annotated for X and Y data flows.

Figure 3.10: Control flow graph annotated for Z dataflow.

Figure 3.11: Control flow graph annotated for V dataflow.

STRATEGIES OF DATA FLOW TESTING:

- **INTRODUCTION:**

  - Data Flow Testing Strategies are structuralstrategies.
  - In contrast to the path-testing strategies, data-flow strategies take into account what happens to data objects on the links in addition to the raw connectivity of the graph.
  - In other words, data flow strategies require data-flow link weights(d,k,u,c,p).
  - Data Flow Testing Strategies are based on selecting test path segments (also called **sub paths**) that satisfy some characteristic ofdata flows for all dataobjects.
  - For example, all sub paths that contain a d (or u, k, du,dk).
  - A strategy X is **stronger** than another strategy Y if all test cases produced under Y are included in those produced under X - conversely for**weaker**.

- TERMINOLOGY:
  1. **Definition-Clear Path Segment**, with respect to variable X, is a connected sequence of links such that X is (possibly) defined on the first link and not redefined or killed on any subsequent link of that path segment. ll paths inFigure 3.9 are definition clear because variables X and Y are defined only on the first link (1,3) and not thereafter. In Figure 3.10, we have a more complicated situation. The following path segments are definition-clear: (1,3,4), (1,3,5), (5,6,7,4), (7,8,9,6,7), (7,8,9,10), (7,8,10), (7,8,10,11). Subpath (1,3,4,5) is not definition-clear because the variable is defined on (1,3) and again on (4,5). For practice, try finding all the definition-clear subpaths for this routine (i.e., for all variables).
  2. **Loop-Free Path Segment** is a path segment for which every node in it is visited atmost once. For Example, path (4,5,6,7,8,10) in Figure 3.10 is loop free, but path (10,11,4,5,6,7,8,10,11,12) is not because nodes 10 and 11 are each visitedtwice.
  3. **Simple path segment** is a path segment in which at most one node is visited twice. For example, in Figure 3.10, (7,4,5,6,7) is a simple path segment. A simple path segment is either loop-free or if there is a loop, only one node isinvolved.
  4. A **du path** from node i to k is a path segment such that if the last link has a computational use of X, then the path is simple and definition-clear; if the penultimate (last but one) node is j - that is, the path is (i,p,q,...,r,s,t,j,k) and link (j,k) has a predicate use - then the path from i to j is both loop-free and definition-clear.

STRATEGIES: The structural test strategies discussed below are based on the program's control flow graph. They differ in the extent to which predicate uses and/or computational uses of variables are included in the test set. Various types of data flow testing strategies in decreasing order of their effectivenessare:

All - du Paths (ADUP): The all-du-paths (ADUP) strategy is the strongest data-flow testing strategy discussed here. It requires that every du path from every definition of every variable to every some test.

For variable X and Y:In Figure 3.9, because variables X and Y are used only on link (1,3), any test that starts at the entry satisfies this criterion (for variables X and Y, but not for all variables as required by the strategy).

For variable Z: The situation for variable Z (Figure 3.10) is more complicated because the variable is redefined in many places. For the definition on link (1,3) we must exercise paths that include subpaths (1,3,4) and (1,3,5). The definition on link (4,5) is covered by any path that includes (5,6), such as subpath (1,3,4,5,6, ...). The (5,6) definition requires paths that include subpaths (5,6,7,4) and (5,6,7,8).

For variable V: Variable V (Figure 3.11) is defined only once on link (1,3). Because V has a predicate use at node 12 and the subsequent path to the end must be forced for both directions at node 12, the all-du-paths strategy for this variable requires that we exercise all loop-free entry/exit paths and at least one path that includes the loop caused by (11,4).

Note that we must test paths that include both subpaths (3,4,5) and (3,5) even though neither of these has V definitions. They must be included because they provide alternate du paths to the V use on link (5,6). Although (7,4) is not used in the test set for variable V, it will be included in the test set that covers the predicate uses of array variable V() and U.

The all-du-paths strategy is a strong criterion, but it does not take as many tests as it might seem at first because any one test simultaneously satisfies the criterion for several definitions and uses of several different variables.

All Uses Startegy (AU): The all uses strategy is that at least one definition clear path from every definition of every variable to every use of that definition be exercised under some test.

Just as we reduced our ambitions by stepping down from all paths (P) to branch coverage (C2), say, we can reduce the number of test cases by asking that the test set should include at least one path segment from every definition to every use that can be reached by that definition.

For variable V: In Figure 3.11, ADUP requires that we include subpaths (3,4,5) and (3,5) in some test because subsequent uses of V, such as on link (5,6), can be reached by either alternative. In AU either (3,4,5) or (3,5) can be used to start paths, but we don't have to use both. Similarly, we can skip the (8,10) link if we've included the (8,9,10) subpath.

Note the hole. We must include (8,9,10) in some test cases because that's the only way to reach the c use at link (9,10) - but suppose our bug for variable V is on link (8,10) after all? Find a covering set of paths under AU for Figure 3.11.

All p-uses/some c-uses strategy (APU+C) : For every variable and every definition of that variable, include at least one definition free path from the definition to every predicate use; if there are definitions of the variables that are not covered by the above prescription, then add computational use test cases as required to cover every definition.

For variable Z:In Figure 3.10, for APU+C we can select paths that all take the upper link (12,13) and therefore we do not cover the c-use of Z: but that's okay according to the strategy's definition because every definition is covered.

Links (1,3), (4,5), (5,6), and (7,8) must be included because they contain definitions for variable Z. Links (3,4), (3,5), (8,9), (8,10), (9,6), and (9,10) must be included because they contain predicate uses of Z. Find a covering set of test cases under APU+C for all variables inthis example - it only takes two tests.

For variable V:In Figure 3.11, APU+C is achieved for V by (1,3,5,6,7,8,10,11,4,5,6,7,8,10,11,12[upper], 13,2) and (1,3,5,6,7,8,10,11,12[lower], 13,2). Note that the c-use at (9,10) need not be included under the APU+C criterion.

All c-uses/some p-uses strategy (ACU+P) : The all c-uses/some p-uses strategy (ACU+P) is to first ensure coverage by computational use cases and if any definition is not covered by the previously selected paths, add such predicate use cases as are needed to assure that every definition is included in some test.

For variable Z: In Figure 3.10, ACU+P coverage is achieved for Z by path (1,3,4,5,6,7,8,10, 11,12,13[lower], 2), but the predicate uses of several definitions are not covered. Specifically, the (1,3) definition is not covered for the (3,5) p-use, the (7,8) definition is not covered for the (8,9), (9,6) and (9, 10) p-uses.

The above examples imply that APU+C is stronger than branch coverage but ACU+P may be weaker than, or incomparable to, branch coverage.

All Definitions Strategy (AD) : The all definitions strategy asks only every definition of every variable be covered by atleast one use of that variable, be that use a computational use or a predicate use.

For variable Z: Path (1,3,4,5,6,7,8, . . .) satisfies this criterion for variable Z, whereas any entry/exit path satisfies it for variable V.
From the definition of this strategy we would expect it to be weaker than both ACU+P and APU+C.

1. **All Predicate Uses (APU), All Computational Uses (ACU) Strategies :** The all predicate uses strategy is derived from APU+C strategy by dropping the requirement that we include a c-use for the variable if there are no p-uses for the variable. The all computational uses strategy is derived from ACU+P strategy by dropping the requirement that we include a p-use for the variable if there are no c-uses for thevariable.

It is intuitively obvious that ACU should be weaker than ACU+P and that APU should be weaker thanAPU+C.

ORDERING THE STRATEGIES:

Figure 3.12compares path-flow and data-flow testing strategies. The arrows denote that the strategy at the arrow's tail is stronger than the strategy at the arrow's head

Figure 3.12: Relative Strength of Structural Test Strategies.

- The right-hand side of this graph, along the path from "all paths" to "all statements" is the more interesting hierarchy for practicalapplications.
- Note that although ACU+P is stronger than ACU, both are incomparable to the predicate-biased strategies. Note also that "all definitions" is not comparable to ACU orAPU.

SLICING AND DICING:

- A (static) program **slice** is a part of a program (e.g., a selected set of statements) defined with respect to a given variable X (where X is a simple variable or a data vector) and a statement i: it is the set of all statements that could (potentially, under static analysis) affect the value of X at statement i - where the influence ofa faulty statement could result from an improper computational use or predicate use of some other variables at priorstatements.
- If X is incorrect at statement i, it follows that the bug must be in the program slice for X with respect toi
- A program **dice** is a part of a slice in which all statements which are known to be correct have beenremoved.
- In other words, a dice is obtained from a slice by incorporating information obtained through testing or experiment (e.g.,debugging).

- The debugger first limits her scope to those prior statements that could have caused the faulty value at statement i (the slice) and then eliminates from further consideration those statements that testing has shown to becorrect.
- Debugging can be modeled as an iterative procedure in which slices are further refined by dicing, where the dicing information is obtained from ad hoc tests aimed primarily at eliminating possibilities. Debugging ends when the dice has been reduced to the one faultystatement.

- **Dynamic slicing** is a refinement of static slicing in which only statements on achievable paths to the statement in question areincluded.

PART B(10 MARKS)

1. Demonstrate the strategies of data flow testing with neatdiagrams.
2. Differentiate various transaction flow testingtechniques
3. Write a short note on slicing anddicing
4. Report different types of data flowmachines
5. Illustrate the usage of transactionflows

UNIT III DOMAIN TESTING

DOMAINS AND PATHS:

- **INTRODUCTION:**
  - **Domain:**Inmathematics, domainisasetofpossiblevaluesofanindependent variable or the variables of afunction.
  - Programsas inputdataclassifiers:domaintestingattemptstodeterminewhether the classification is or is notcorrect.
  - Domain testing can be based on specifications or equivalentimplementation information.
  - If domain testing is based on specifications, it is a functional testtechnique.
  - If domain testing is based implementation details, it is a structural testtechnique.
  - For example, you're doing domain testing when you check extreme values ofan inputvariable.

  All inputs to a program can be considered as if they are numbers. For example, a character string can be treated as a number by concatenating bits and looking at them as if they were a binary integer. This is the view in domain testing, which is why this strategy has a mathematicalflavor.

- **THE MODEL:** The following figure is a schematic representation of domaintesting.

Figure 4.1: Schematic Representation of Domain Testing.

  - Before doing whatever it does, a routine must classify the input and set it moving on the rightpath.
  - An invalid input (e.g., value too big) is just a special processing case called 'reject'.
  - The input then passes to a hypothetical subroutine rather than oncalculations.
  - Indomaintesting,wefocusontheclassificationaspect oftheroutine ratherthan on thecalculations.
  - Structural knowledge is not needed for this model - only a consistent,complete specification of input values for eachcase.
  - We can infer that for each case there must be at least one path to process thatcase.

- A DOMAIN IS ASET:
    - An input domain is aset.
    - If the source language supports set definitions (E.g. PASCAL set types and C enumerated types) less testing is needed because the compiler does much of it for us.
    - Domain testing does not work well with arbitrary discrete sets ofdataobjects.
    - Domain for a loop-free program corresponds to a set of numbers defined over the inputvector.

- DOMAINS, PATHS ANDPREDICATES:
    - In domain testing, predicates are assumed to be interpreted in terms of input vectorvariables.
    - If domain testing is applied to structure, then predicate interpretation must be based on actual paths through the routine - that is, based on the implementation control flowgraph.
    - Conversely, if domain testing is applied to specifications, interpretation is based on a specified data flow graph for the routine; but usually, as is the nature of specifications, no interpretation is needed because the domains are specified directly.
    - For every domain, there is at least one path through theroutine.
    - There may be more than one path if the domain consists of disconnected parts or if the domain is defined by the union of two or moredomains.
    - Domains are defined their boundaries. Domain boundaries are also where most domain bugsoccur.
    - For every boundary there is at least one predicate that specifies what numbers belong to the domain and what numbersdon't.
      For example, in the statement IF x>0 THEN ALPHA ELSE BETA we know that numbers greater than zero belong to ALPHA processing domain(s) while zero and smaller numbers belong to BETA domain(s).
    - A domain may have one or more boundaries - no matter how many variables define it. For example, if the predicate is x2 + y2 < 16, the domain is the inside of a circle of radius 4 about the origin. Similarly, we could define a spherical domain with one boundary but in threevariables.
    - Domains are usually defined by many boundary segments and therefore by many predicates. i.e. the set of interpreted predicates traversed on that path (i.e., the path's predicate expression) defines the domain'sboundaries.

- A DOMAINCLOSURE:
    - A domain boundary is **closed** with respect to a domain if the points onthe boundary belong to thedomain.
    - If the boundary points belong to some other domain, the boundary is said tobe open.
    - Figure 4.2 shows three situations for a one-dimensional domain - i.e., a domain defined over one input variable; call itx

The importance of domain closure is that incorrect closure bugs are frequent domain bugs. For example, x >= 0 when x > 0 was intended

Figure 4.2: Open and Closed Domains.

- **DOMAIN DIMENSIONALITY:**
  - Every input variable adds one dimension to thedomain.
  - One variable defines domains on a numberline.
  - Two variables defineplanardomains.
  - Three variables definesoliddomains.
  - Every new predicate slices through previously defined domains and cutsthemin half.
  - Every boundary slices through the input vector space with a dimensionalitywhich is less than the dimensionality of thespace.
  - Thus, planes are cut by lines and points, volumes by planes, lines and pointsand n-spaces by hyperplanes.

- BUGASSUMPTION:
  - The bug assumption for the domain testing is that processing is okay but the domain definition iswrong.
  - An incorrectly implemented domain means that boundaries are wrong, whichmay in turn mean that control flow predicates arewrong.
  - Many different bugs can result in domain errors. Some ofthemare:

    Domain Errors:
    - **Double Zero Representation: In** computers or Languages that have a distinct positive and negative zero, boundary errors for negative zero are common.
    - **Floating point zero check:** A floating point number can equal zero only if the previous definition of that number set it to zero or if it is subtracted from itself or multiplied by zero. So the floating point zero check to be done against an epsilonvalue.
    - **Contradictory domains:** An implemented domain can never be ambiguous or contradictory, but a specified domain can. Acontradictory

domain specification means that at least two supposedly distinct domains overlap.

- **Ambiguous domains:** Ambiguous domains means that union of the domains is incomplete. That is there are missing domains or holes in the specified domains. Not specifying what happens to points on the domain boundary is a commonambiguity.

- **Over specified Domains:** his domain can be overloaded with so many conditions that the result is a null domain. Another way to put it is to say that the domain's path isunachievable.

- **Boundary Errors:** Errors caused in and around the boundary of adomain. Example, boundary closure bug, shifted, tilted, missing, extraboundary.

- **Closure Reversal:** A common bug. The predicate is defined in termsof >=. The programmer chooses to implement the logical complement and incorrectly uses <= for the new predicate; i.e., x >= 0 is incorrectly negated as x <= 0, thereby shifting boundary values to adjacentdomains.

- **Faulty Logic:** Compound predicates (especially) are subject to faulty logic transformations and improper simplification. If the predicates define domain boundaries, all kinds of domain bugs can result from faulty logic manipulations.

- **RESTRICTIONS TO DOMAIN TESTING:** Domain testing has restrictions, asdo other testing techniques. Some of theminclude:

  - **Co-incidental Correctness:** Domain testing isn't good at finding bugs for which the outcome is correct for the wrong reasons. If we're plagued by coincidental correctness we may misjudge an incorrect boundary. Note that this implies weakness for domain testing when dealing with routines that have binary outcomes (i.e.,TRUE/FALSE)

  - **Representative Outcome:** Domain testing is an example of **partition testing**. Partition-testing strategies divide the program's input space into domains such that all inputs within a domain are equivalent (not equal, but equivalent) in the sense that any input represents all inputs in thatdomain.
  - If the selected input is shown to be correct by a test, then processing is presumed correct, and therefore all inputs within that domain are expected (perhaps unjustifiably) to be correct. Most test techniques, functional or structural, fall under partition testing and therefore make this representative outcomeassumption. For example, x^2 and 2^x are equal for x = 2, but the functions are different. The functional differences between adjacent domains are usually simple, such as x + 7 versus x + 9, rather than x^2 versus2^x.

Simple Domain Boundaries and Compound Predicates: Compound predicates in which each part of the predicate specifies a different boundary are not a problem: for example,x >= 0 AND x < 17, just specifies two domain boundaries by one compound predicate. As

an example of a compound predicate that specifies one boundary, consider: x = 0 AND y >= 7 AND y <= 14. This predicate specifies one boundary equation (x = 0) but alternates closure, putting it in one or the other domain depending on whether y < 7 or y > 14. Treat compound predicates with respect because they're more complicated than they seem.

- **Functional Homogeneity of Bugs:** Whatever the bug is, it will not change the functional form of the boundary predicate. For example, if the predicate is ax >= b, the bug will be in the value of a or b but it will not change the predicate toax >= b, say.

- **Linear Vector Space:** Most papers on domain testing, assume linear boundaries - not a bad assumption because in practice most boundary predicates arelinear.

- **Loop Free Software:** Loops are problematic for domain testing. The trouble with loops is that each iteration can result in a different predicate expression (after interpretation), which means a possible domain boundarychange.

NICE AND UGLY DOMAINS:

- **NICEDOMAINS:**
  - **Where do these domains comefrom?**
    Domains are and will be defined by an imperfect iterative process aimedat achieving (user, buyer, voter)satisfaction.
  - Implemented domains can't be incomplete or inconsistent. Every input will be processed (rejection is a process), possibly forever. Inconsistent domains will be made consistent.
  - Conversely, specified domains can be incomplete and/or inconsistent. Incomplete in this context means that there are input vectors for which no path is specified, and inconsistent means that there are at least two contradictory specifications over the same segment of the inputspace.
  - Some important properties of nice domains are: **Linear, Complete, Systematic, And Orthogonal, Consistently closed, Convex and simplyconnected.**
  - To the extent that domains have these properties domain testing is easy as testing gets.
  - The bug frequency is lesser for nice domain than for uglydomains.

Figure 4.3: Nice Two-Dimensional Domains.

- **LINEAR AND NON LINEARBOUNDARIES:**
  - Nice domain boundaries are defined by linear inequalities orequations.
  - The impact on testing stems from the fact that it takes only two points to determine a straight line and three points to determine a plane and in general n+ 1 point to determine an n-dimensional hyperplane.
  - In practice more than 99.99% of all boundary predicates are either linear or can be linearized by simple variabletransformations.

- COMPLETEBOUNDARIES:
  - Nice domain boundaries are complete in that they span the number space from plus to minus infinity in alldimensions.
  - Figure 4.4 shows some incomplete boundaries. Boundaries A and E havegaps.
  - Such boundaries can come about because the path that hypothetically corresponds to them is unachievable, because inputs are constrained in such a way that such values can't exist, because of compound predicates that define a single boundary, or because redundant predicates convert such boundary values into a nullset.
  - The advantage of complete boundaries is that one set of tests is needed to confirm the boundary no matter how many domains itbounds.
  - If the boundary is chopped up and has holes in it, then every segment of that boundary must be tested for every domain itbounds.

Figure 4.4: Incomplete Domain Boundaries.

- **SYSTEMATIC BOUNDARIES:**
  - Systematicboundarymeansthatboundaryinequalitiesrelatedbyasimple function such as aconstant.

In Figure 4.3 for example, the domain boundaries for u and v differ only by a constant.

$$f_1(X) >= k_1 \text{ or } f_1(X) >= g(1,c)$$
$$f_1(X) >= k_2 \quad f_2(X) >= g(2,c)$$
$$\ldots\ldots\ldots\ldots \quad \ldots\ldots\ldots\ldots$$
$$f_i(X) >= k_i \quad f_i(X) >= g(i,c)$$

- where f_i is an arbitrary linear function, X is the input vector, k_i and c are constants, and $g(i,c)$ is a decent function over i and c that yields a constant, such as $k + ic$.
- The first example is a set of parallel lines, and the second example is a set of systematically (e.g., equally) spaced parallel lines (such as the spokes of a wheel, if equally spaced in angles,systematic).
- If the boundaries are systematic and if you have one tied down and generate tests for it, the tests for the rest of the boundaries in that set can be automatically generated.

- ORTHOGONALBOUNDARIES:
  - Two boundary sets U and V (See Figure 4.3) are said to be orthogonal if every inequality in V is perpendicular to every inequality inU.
  - If two boundary sets are orthogonal, then they can be testedindependently
  - In Figure 4.3 we have six boundaries in U and four in V. We can confirm the boundary properties in a number of tests proportional to 6 + 4 = 10 (O(n)). If we tilt the boundaries to get Figure4.5,
  - we must now test the intersections. We've gone from a linear number of cases to a quadratic: from O(n) to$O(n^2)$.

Figure 4.5: Tilted Boundaries.

Figure 4.6: Linear, Non-orthogonal Domain Boundaries.

- Actually, there are two different but related orthogonality conditions. Sets of boundaries can be orthogonal to one another but not orthogonal to the coordinate axes (condition 1), or boundaries can be orthogonal to the coordinate axes (condition2).

- CLOSURECONSISTENCY:
  - Figure 4.6 shows another desirable domain property: boundary closures are consistent andsystematic.
  - The shaded areas on the boundary denote that the boundary belongs to the domain in which the shading lies - e.g., the boundary lines belong to the domains on the right.
  - Consistent closure means that there is a simple pattern to the closures - for example, using the same relational operator for all boundaries of a set of parallel boundaries.

- CONVEX:
  - A geometric figure (in any number of dimensions) is convex if you can take two arbitrary points on any two different boundaries, join them by a line and all points on that line lie within thefigure.
  - Nice domains are convex; dirty domainsaren't.
  - You can smell a suspected concavity when you see phrases such as: ". . . exceptif . . .," "However . . .," ". . . but not. " In programming, it's often the buts in the specification that kill you.

- SIMPLY CONNECTED:
  - Nice domains are simply connected; that is, they are in one piece rather than pieces all over the place interspersed with otherdomains.
  - Simple connectivity is a weaker requirement than convexity; if a domain is convex it is simply connected, but not viceversa.
  - Consider domain boundaries defined by a compound predicate of the (Boolean) form ABC. Say that the input space is divided into two domains, one defined by ABC and, therefore, the other defined by itsnegation.
  - For example, suppose we define valid numbers as those lying between 10 and 17 inclusive. The invalid numbers are the disconnected domain consisting of numbers less than 10 and greater than17.
  - Simple connectivity, especially for default cases, may beimpossible.

- UGLY DOMAINS:
  - Some domains are born ugly and some are uglified by badspecifications.
  - Every simplification of ugly domains by programmers can be either goodorbad.
  - Programmers in search of nice solutions will "simplify" essential complexity out of existence. Testers in search of brilliant insights will be blind to essential complexity and therefore miss importantcases.
  - If the ugliness results from bad specifications and the programmer's simplification is harmless, then the programmer has made uglygood.

- But if the domain's complexity is essential (e.g., the income tax code), such "simplifications" constitutebugs.
- Nonlinear boundaries are so rare in ordinary programming that there's no information on how programmers might "correct" such boundaries if they're essential.

- AMBIGUITIES ANDCONTRADICTIONS:
  - Domain ambiguities are holes in the inputspace.
  - The holes may lie within the domains or in cracks betweendomains.
  - Two kinds ofcontradictions are possible: overlapped domain specifications and overlapped closurespecifications
  - Figure 4.7c shows overlapped domains and Figure 4.7d shows dualclosure assignment.

Figure 4.7: Domain Ambiguities and Contradictions.

- **SIMPLIFYING THETOPOLOGY:**
  - The programmer's and tester's reaction to complex domains is the same -simplify
  - There are three generic cases: **concavities**, **holes** and **disconnectedpieces**.
  - Programmers introduce bugs and testers misdesign test cases by: smoothing out concavities (Figure 4.8a), filling in holes (Figure 4.8b), and joining disconnected pieces (Figure4.8c).

Figure 4.8: Simplifying the topology.

- **RECTIFYING BOUNDARYCLOSURES:**
  - If domain boundaries are parallel but have closures that go every which way (left, right, left . . .) the natural reaction is to make closures go the same way (see Figure4.9).

Figure 4.9: Forcing Closure Consistency.

DOMAIN TESTING:

- **DOMAIN TESTING STRATEGY:** The domain-testing strategy is simple, although possibly tedious(slow).
  - Domains are defined by their boundaries; therefore, domain testing concentrates test points on or nearboundaries.
  - Classify what can go wrong with boundaries, then define a test strategy for each case. Pick enough points to test for all recognized kinds of boundaryerrors.
  - Because every boundary serves at least two different domains, test points used to check one domain can also be used to check adjacent domains. Remove redundant testpoints.
  - Run the tests and by posttest analysis (the tedious part) determine if any boundaries are faulty and if so,how.
  - Run enough tests to verify every boundary ofeverydomain.

- DOMAIN BUGS AND HOW TO TEST FORTHEM:
  - An **interior point** (Figure 4.10) is a point in the domain such that all points within an arbitrarily small distance (called an epsilon neighborhood) are also in the domain.
  - A **boundary point** is one such that within an epsilon neighborhood there are points both in the domain and not in thedomain.
  - An **extreme point** is a point that does not lie between any two other arbitrary but distinct points of a (convex)domain.

Figure 4.10: Interior, Boundary and Extreme points.

- An **on point** is a point on theboundary.
- If the domain boundary is closed, an **off point** is a point near the boundary but in the adjacentdomain.
- If the boundary is open, an off point is a point near the boundary but in the domain being tested; see Figure 4.11. You can remember this by the acronym COOOOI: Closed Off Outside, Open OffInside.

Figure 4.11: On points and Off points.

- Figure 4.12 shows generic domain bugs: closure bug, shifted boundaries, tilted boundaries, extra boundary, missingboundary.

Figure 4.12: Generic Domain Bugs.

TESTING ONE DIMENSIONAL DOMAIN:

The closure can be wrong (i.e., assigned to the wrong domain) or the boundary (a point in this case) can be shifted one way or the other, we can be missing a boundary, or we can have an extra boundary.

1. Figure 4.13 shows possible domain bugs for a one-dimensional open domain boundary.
2. In Figure 4.13a we assumed that the boundary was to be open for A. The bug we're looking for is a closure error, which converts > to >= or < to <= (Figure 4.13b). One test (marked x) on the boundary point detects this bug because processing for that point will go to domain A rather thanB.
3. In Figure 4.13c we've suffered a boundary shift to the left. The test point we used for closure detects this bug because the bug forces the point from the B domain, where it should be, to A processing. Note that we can't distinguish between a shift and a closure error, but we do know that we have abug.

Figure 4.13: One Dimensional Domain Bugs, Open Boundaries.

4. Figure 4.13d shows a shift the other way. The on point doesn't tell us anything because the boundary shift doesn't change the fact that the test point will be processed in B. To detect this shift we need a point close to the boundary but within A. The boundary is open, therefore by definition, the off point is in A (Open OffInside).
5. The same open off point also suffices to detect a missing boundary because what should have been processed in A is now processed inB.
6. To detect an extra boundary we have to look at two domain boundaries. In this context an extra boundary means that A has been split in two. The two off points that we selected before (one for each boundary) does the job. If point C had been a closed boundary, the on test point at C would doit.
7. For closed domains look at Figure 4.14. As for the open boundary, a test point on the boundary detects the closure bug. The rest of the cases are similar to the open boundary, except now the strategy requires off points just outside thedomain.

Figure 4.14: One Dimensional Domain Bugs, Closed Boundaries.

- **TESTING TWO DIMENSIONALDOMAINS:**

1. Figure 4.15 shows possible domain boundary bugs for a two-dimensionaldomain.
2. A and B are adjacent domains and the boundary is closed with respect to A, which means that it is open with respect toB.

Figure 4.15: Two Dimensional Domain Bugs.

3. **For ClosedBoundaries:**
   Closure Bug: Figure 4.15a shows a faulty closure, such as might be caused by using a wrong operator (for example, x >= k when x > k was intended, or vice

versa). The two on points detect this bug because those values will get B rather than A processing.

1. **Shifted Boundary:** In Figure 4.15b the bug is a shift up, which converts part of domain B into A processing, denoted by A'. This result is caused by an incorrect constant in a predicate, such as x + y >= 17 when x + y >= 7 was intended. The off point (closed off outside) catches this bug. Figure 4.15c shows a shift down that is caught by the two onpoints.
2. **Tilted Boundary:** A tilted boundary occurs when coefficients in the boundary inequality are wrong. For example, 3x + 7y > 17 when 7x +3y> 17 was intended. Figure 4.15d has a tilted boundary, which creates erroneous domain segments A' and B'. In this example the bug is caught by the left onpoint.
3. **Extra Boundary:** An extra boundary is created by an extra predicate. An extra boundary will slice through many different domains and will therefore cause many test failures for the same bug. The extra boundary in Figure 4.15e is caught by two on points, and depending on which way the extra boundary goes, possibly by the off pointalso.
4. **Missing Boundary:** A missing boundary is created by leaving a boundary predicate out. A missing boundary will merge different domains and will cause many test failures although there is only one bug. A missing boundary, shown in Figure 4.15f, is caught by the two on points because the processing for A and B is the same - either A or Bprocessing.

- **PROCEDURE FOR TESTING:** The procedure is conceptually is straight forward. It can be done by hand for two dimensions and for a few domains and practically impossible for more than twovariables.
    1. Identify inputvariables.
    2. Identify variable which appear in domain defining predicates, such as control flow predicates.
    3. Interpret all domain predicates in terms of inputvariables.
    4. For p binary predicates, there are at most 2^p combinations of TRUE-FALSE values and therefore, at most 2^p domains. Find the set of all non null domains. The result is a boolean expression in the predicates consisting a set of AND terms joined by OR's. For example ABC+DEF+GHI..... Where the capital letters denote predicates. Each product term is a set of linear inequality that defines a domain or a part of a multiply connecteddomains.
    5. Solve these inequalities to find all the extreme points of each domain using any of the linear programmingmethods.

DOMAIN AND INTERFACE TESTING

- **INTRODUCTION:**
    - Recallthatwedefinedintegrationtestingastestingthecorrectnessoftheinterface between two otherwise correctcomponents.

- Components A and B have been demonstrated to satisfy their component tests, and as part of the act of integrating them we want to investigate possible inconsistencies across theirinterface.
- Interface between any two components is considered as a subroutinecall.
- We're looking for bugs in that "call" when we do interfacetesting.
- Let's assume that the call sequence is correct and that there are no type incompatibilities.
- For a single variable, the domain span is the set of numbers between (and including) the smallest value and the largest value. For every input variable we want (at least): compatible domain spans and compatible closures (Compatible but need not beEqual).

- DOMAINS ANDRANGE:
  - The set of output values produced by a function is called the **range** of the function, in contrast with the **domain**, which is the set of input values over which the function isdefined.
  - For most testing, our aim has been to specify input values and to predict and/or confirm output values that result from thoseinputs.
  - Interface testing requires that we select the output values of the calling routine*i.e.* caller's range must be compatible with the called routine's domain.
  - An interface test consists of exploring the correctness of the followingmappings:

    caller domain -->callerrange (caller unittest)
    caller range -->called domain (integration test)
    called domain -->calledrange (called unittest)

- CLOSURECOMPATIBILITY:
  - Assume that the caller's range and the called domain spans the same numbers - for example, 0 to17.
  - Figure 4.16 shows the four ways in which the caller's range closure and the called's domain closure canagree.
  - The thick line means closed and the thin line means open. Figure 4.16 shows the four cases consisting of domains that are closed both on top (17) and bottom (0), open top and closed bottom, closed top and open bottom, and open top and bottom.

Figure 4.16: Range / Domain Closure Compatibility.

  - Figure 4.17 shows the twelve different ways the caller and the called can disagree about closure. Not all of them are necessarily bugs. The four cases inwhicha

caller boundary is open and the called is closed (marked with a "?") are probably not buggy. It means that the caller will not supply such values but the called can accept them.

Figure 4.17: Equal-Span Range / Domain Compatibility Bugs.

- **SPAN COMPATIBILITY:**
  - Figure 4.18 shows three possibly harmless spanincompatibilities.

Figure 4.18: Harmless Range / Domain Span incompatibility bug (Caller Span is smaller than Called).

  - In all cases, the caller's range is a subset of the called's domain. That's not necessarily abug.
  - The routine is used by many callers; some require values inside a range and some don't. This kind of span incompatibility is a bug only if the caller expects the called routine to validate the called number for thecaller.
  - Figure 4.19a shows the opposite situation, in which the called routine's domain has a smaller span than the caller expects. All of these examples arebuggy.

Figure 4.19: Buggy Range / Domain Mismatches

- In Figure 4.19b the ranges and domains don't line up; hence good values are rejected, bad values are accepted, and if the called routine isn't robust enough, we have crashes.
- Figure 4.19c combines these notions to show various ways we can have holes in the domain: these are all probablybuggy.

- INTERFACE RANGE / DOMAIN COMPATIBILITYTESTING:
  - For interface testing, bugs are more likely to concern single variables rather than peculiar combinations of two or morevariables.
  - Test every input variable independently of other input variables to confirm compatibility of the caller's range and the called routine's domain span and closure of every domain defined for thatvariable.
  - There are two boundaries to test and it's a one-dimensional domain; therefore, it requires one on and one off point per boundary or a total of two on points and two off points for the domain - pick the off points appropriate to the closure (COOOOI).
  - Start with the called routine's domains and generate test points in accordance to the domain-testing strategy used for that routine in componenttesting.
  - Unless you're a mathematical whiz you won't be able to do this without tools for more than one variable at atime.

PART A(2 MARKS)

1. Define UnitTesting

A unit is the smallest possible testable software component that can be characterized in several ways.

2. List the components suitable for unit test.

Procedures andfunctions

Classes/objects and methods

Procedure-sized reusable components

3. What is Testharness?

The auxiliary code developed to support to testing of units and components is called atest harness. The harness consists of drivers that call the target code and stubs that represent modules it calls

4. What is the advantage of Bottom upintegration?

Bottom-up integration has the advantage that the lower-level modules are usually well tested early in the integration process. This is important if these modules are candidates for reuse

5. What is acluster?

A cluster consists of classes that are related, for example, they may work together to support a required functionality for the complete system.

6. DefineLoad.

A load is a series of inputs that simulates a group of transactions.

7. What is meant by Stress testing?

When a system is tested with a load that causes it to allocate its resources in maximum amounts, this is called stress testing

8. Define Recovery testing. Recovery testing subjects a system to losses of resources in order to determine if it can recover properly from these losses**.**

PART B(10 MARKS)

1. Calculate the maximum path count and lower path count for the following flow graph with path expression: a(b+c)d{e(fi)*fgj(m+l)k}*e(fi)*fgh. Each link is given a weight of 1. Outer loop will take exactly four times and inner loop takes zero or three times its pathexpression

2. Demonstrate by means of truth tables the validity of the following theorems of Boolean algebra: (i) Associate laws. (ii) Demorgan's theorem for three variables. (iii) Distributivelaw. (iv) Absorption rule

3. Explain about Maximum path count arithmetic with anexample?

4. Define structured code. Explain lower path countArithmetic?

5. Demonstrate the steps involved in node reduction procedure. Illustrate all the steps with help of neat labeleddiagram

UNIT IV

PATHS, PATH PRODUCTS AND REGULAREXPRESSIONS

PATH PRODUCTS AND PATH EXPRESSION:

- **MOTIVATION:**
  - Flow graphs are being an abstract representation ofprograms.
  - Any question about a program can be cast into an equivalent question aboutan appropriate flowgraph.
  - Most software development, testing and debugging tools use flow graphs analysis techniques.

- PATHPRODUCTS:
  - Normally flow graphs used to denote only control flowconnectivity.
  - The simplest weight we can give to a link is aname.
  - Using link names as weights, we then convert the graphical flow graph into an equivalent algebraic like expressions which denotes the set of all possible paths from entry to exit for the flow graph.
  - Every link of a graph can be given aname.
  - The link name will be denoted by lower case italic letters In tracing a path or path segment through a flow graph, you traverse a succession of linknames.
  - The name of the path or path segment that corresponds to those links is expressed naturally by concatenating those linknames.
  - For example, if you traverse links a,b,c and d along some path, the name for that path segment is abcd. This path name is also called a **path product.** Figure 5.1 shows someexamples:

Figure 5.1: Examples of paths.

- **PATHEXPRESSION:**
  - Consider a pair of nodes in a graph and the set of paths between thosenode.
  - Denotethat setofpathsbyUppercaselettersuchasX,Y.FromFigure 5.1c,the members of the path set can be listed asfollows:

    ac, abc, abbc, abbbc, abbbbc.....
  - Alternatively, the same set of paths can be denoted by:

    ac+abc+abbc+abbbc+abbbbc+.....
  - The+signisunderstoodtomean"or"betweenthetwonodesofinterest,pathsac, or abc, or abbc, and so on can betaken.
  - Any expression that consists of path names and "OR"s and which denotes a set of paths between two nodes is called a "**PathExpression".**

- PATHPRODUCTS:
  - The name of a path that consists of two successive path segmentsisconveniently expressed by the concatenation or **Path Product** of the segmentnames.
  - For example, if X and Y are defined as X=abcde,Y=fghij,then thepath corresponding to X followed by Y is denotedby

    XY=abcdefghij
  - Similarly,
    YX=fghijabcde
    aX=aabcde
    Xa=abcdea
    XaX=abcdeaabcde
  - If X and Y represent sets of paths or path expressions, their product representsthe set of paths that can be obtained by following every element of X by any element of Y in all possible ways. Forexample,
  - X = abc + def +ghi
  - Y = uvw + z

    Then,
    XY = abcuvw + defuvw + ghiuvw + abcz + defz + ghiz
  - If a link or segment name is repeated, that fact is denoted by an exponent. The exponent's value denotes the number ofrepetitions:
  - $a^1 = a$; $a^2 = aa$; $a^3 = aaa$; $a^n = aaaa$ . .

    . n times. Similarly, if X = abcdethen

    X^1 = abcde
    X^2 = abcdeabcde = $(abcde)^2$
    X^3 = abcdeabcdeabcde = $(abcde)^2$abcde
    = abcde$(abcde)^2$ = $(abcde)^3$
  - The path product is not commutative (that isXY!=YX).
  - The path product isAssociative.
    RULE 1: A(BC)=(AB)C=ABC
    where A,B,C are path names, set of path names or path expressions.
  - The zeroth power of a link name, path product, or path expression is also needed for completeness. It is denoted by the numeral "1" and denotes the "path" whose length is zero - that is, the path that doesn't have anylinks.

- $a^0 = 1$
- $X^0 = 1$

- PATHSUMS:
  - The "+" sign was used to denote the fact that path names were part of the same set ofpaths.
  - The "PATH SUM" denotes paths in parallel betweennodes.
  - Links a and b in Figure 5.1a are parallel paths and are denoted by a + b. Similarly, links c and d are parallel paths between the next two nodes and are denoted by c + d.
  - The set of all paths between nodes 1 and 2 can be thought of as a set of parallel paths and denoted byeacf+eadf+ebcf+ebdf.
  - If X and Y are sets of paths that lie between the same pair of nodes, then X+Y denotes the UNION of those set of paths. For example, in Figure5.2:

Figure 5.2: Examples of path sums.

The first set of parallel paths is denoted by X + Y + d and the second set by U +V + W + h + i + j. The set of all paths in this flowgraph is f(X + Y + d)g(U + V +W + h + i +j)k

  - The path is a set union operation, it is clearly Commutative andAssociative.
  - RULE 2:X+Y=Y+X
  - RULE 3:(X+Y)+Z=X+(Y+Z)=X+Y+Z

- DISTRIBUTIVELAWS:
  - The product and sum operations are distributive, and the ordinary rules of multiplication apply; thatis

    RULE 4: A(B+C)=AB+AC and (B+C)D=BD+CD
  - Applying these rules to the below Figure 5.1ayields
  - e(a+b)(c+d)f=e(ac+ad+bc+bd)f =eacf+eadf+ebcf+ebdf

- ABSORPTIONRULE:
  - If X and Y denote the same set of paths, then the union of these sets is unchanged; consequently,

    RULE 5: X+X=X (Absorption Rule)
  - If a set consists of paths names and a member of that set is added to it, the "new" name, which is already in that set of names, contributes nothing and can be ignored.
  - Forexample,
  - if X=a+aa+abc+abcd+defthen

    X+a = X+aa = X+abc = X+abcd = X+def = X

    It follows that any arbitrary sum of identical path expressions reduces to the same path expression.

- LOOPS:

  Loops can be understood as an infinite set of parallel paths. Say that the loop consists of a single link b. then the set of all paths through that loop point is b0+b1+b2+b3+b4+b5+.....

Figure 5.3: Examples of path loops.

This potentially infinite sum is denoted by b* for an individual link and by X*

Figure 5.4: Another example of path loops.

- The path expression for the above figure is denoted by thenotation: ab*c=ac+abc+abbc+abbbc+.....
- Evidently, aa*=a*a=a+ and XX*=X*X=X+
- It is more convenient to denote the fact that a loop cannot be taken more than a certain, say n, number oftimes.
- A bar is used under the exponent to denote the fact asfollows: $X^{\underline{n}} = X^0+X^1+X^2+X^3+X^4+X^5+ \ldots\ldots\ldots\ldots +X^n$

RULES 6 - 16:

- The following rules can be derived from the previousrules:
- RULE 6: $X^n + X^{\underline{m}} = X^n$ if n>m RULE 6: $X^n + X^{\underline{m}} = X^{\underline{m}}$ if m>n RULE 7: $X^n X^m = X^{\underline{n+m}}$

  RULE 8: $X^n X^* = X^* X^n = X^*$

  RULE 9: $X^n X^+ = X^+ X^n = X^+$

  RULE 10: $X^* X^+ = X^+ X^* = X^+$ RULE 11: 1 + 1 = 1

  RULE 12: 1X = X1 = X

  Following or preceding a set of paths by a path of zero length does not change the set.

  RULE 13: $1^n = 1^n = 1^* = 1^+ = 1$

  No matter how often you traverse a path of zero length,It is a path of zero length.

  RULE 14: $1^+ + 1 = 1^* = 1$

  The null set of paths is denoted by the numeral 0. it obeys the following rules:

  RULE 15: X+0=0+X=X

  RULE 16: 0X=X0=0

  If you block the paths of a graph for or aft by a graph that has no paths , there

won’t be any paths.

REDUCTION PROCEDURE:

- **REDUCTION PROCEDUREALGORITHM:**
  - This section presents a reduction procedure for converting a flowgraph whose links are labeled with names into a path expression that denotes the set of all entry/exit paths in that flowgraph. The procedure is a node-by-node removal algorithm.
  - The steps in Reduction Algorithm are asfollows:
    6. Combine all serial links by multiplying theirpathexpressions.
    7. Combine all parallel links by adding theirpathexpressions.
    8. Remove all self-loops (from any node to itself) by replacing them with a link of the form X*, where X is the path expression of the link in thatloop.

    STEPS 4 - 8 ARE IN THE ALGORIHTM'S LOOP:
    9. Select any node for removal other than the initial or final node. Replace it with a set of equivalent links whose path expressions correspond to all the ways you can form a product of the set of inlinks with the set of outlinks of that node.
    10. Combine any remaining serial links by multiplying their pathexpressions.
    11. Combine all parallel links by adding their pathexpressions.
    12. Remove all self-loops as in step3.
    13. Does the graph consist of a single link between the entry node and the exit node? If yes, then the path expression for that link is a path expression for the original flowgraph; otherwise, return to step4.
  - A flowgraph can have many equivalent path expressions between a given pair of nodes; that is, there are many different ways to generate the set of all paths between two nodes without affecting the content of thatset.
  - The appearance of the path expression depends, in general, on the order in which nodes are removed.

- CROSS-TERM STEP (STEP4):
  - The cross - term step is the fundamental step of the reductionalgorithm.
  - It removes a node, thereby reducing the number of nodes byone.
  - Successive applications of this step eventually get you down to one entry and one exit node. The following diagram shows the situation at an arbitrary node that has been selected forremoval:

  - From the above diagram, one can infer:
  - (a + b)(c + d + e) = ac + ad + + ae + bc + bd +be

- LOOP REMOVALOPERATIONS:
  - There are two ways of looking at the loop-removal operation:

- In the first way, we remove the self-loop and then multiply all outgoing links by Z*.
- In the second way, we split the node into two equivalent nodes, call them A and A' and put in a link between them whose path expression is Z*. Then we remove node A' using steps 4 and 5 to yield outgoing links whose path expressions are Z*X andZ*Y.

- A REDUCTION PROCEDURE -EXAMPLE:
  - Let us see by applying this algorithm to the following graph where we remove several nodes in order; thatis

Figure 5.5: Example Flowgraph for demonstrating reduction procedure.

  - Remove node 10 by applying step 4 and combine by step 5 toyield

  - Remove node 9 by applying step4 and 5 toyield

  - Remove node 7 by steps 4 and 5, asfollows:

- Remove node 8 by steps 4 and 5, toobtain:

- PARALLEL TERM (STEP6):
  Removal of node 8 above led to a pair of parallel links between nodes 4 and 5.combine them to create a path expression for an equivalent link whose path expression is c+gkh; thatis

- LOOP TERM (STEP7):
  Removing node 4 leads to a loop term. The graph has now been replaced with the following equivalent simpler graph:

- Continue the process by applying the loop-removal step asfollows:

- Removing node 5produces:

- Remove the loop at node 6 toyield:

- Remove node 3 toyield

- Removing the loop and then node 6 result in the followingexpression:
  a(bgjf)*b(c+gkh)d((ilhd)*imf(bjgf)*b(c+gkh)d)*(ilhd)*e

- You can practice by applying the algorithm on the following flowgraphsand generate their respective path expressions:

Figure 5.6: Some graphs and their path expressions.

APPLICATIONS:

- The purpose of the node removal algorithm is to present one very generalized concept- the path expression and way of gettingit.
- Every application follows this commonpattern:
  1. Convert the program or graph into a pathexpression.
  2. Identifyapropertyofinterestandderive anappropriatesetof"arithmetic"rules that characterizes theproperty.

Replace the link names by the link weights for the property of interest. The path expression has now been converted to an expression in some algebra, such as

1. Ordinaryalgebra,regularexpressions,orbooleanalgebra.Thisalgebraic expressionsummarizestheproperty ofinterestoverthe setofallpaths.
2. Simplifyor evaluatetheresulting"algebraic"expressiontoanswerthequestion youasked.

- HOW MANY PATHS IN A FLOW GRAPH?
  - The question is not simple. Here are some ways you could askit:
    1. What is the maximum number of different pathspossible?
    2. What is the fewest number of pathspossible?
    3. How many different paths are thereareally?
    4. What is the average number ofpaths?
  - Determining the actual number of different paths is an inherently difficult problem because there could be unachievable paths resulting from correlated and dependentpredicates.
  - If we know both of these numbers (maximum and minimum number of possible paths) we have a good idea of how complete our testingis.
  - Asking for "the average number of paths" ismeaningless.

- MAXIMUM PATH COUNTARITHMETIC:
  - Label each link with a link weight that corresponds to the number of paths that linkrepresents.
  - Also mark each loop with the maximum number of times that loop can be taken. If the answer is infinite, you might as well stop the analysis because it is clear that the maximum number of paths will beinfinite.
  - There are three cases of interest: parallel links, serial links, andloops.

Case	Path expression	Weight expression
Parallels	A+B	W_A+W_B
Series	AB	W_AW_B
Loop	$A^{\underline{n}}$	$\sum_{j=0}^{n} W_A^j$

  - This arithmetic is an ordinary algebra. The weight is the number of paths in each set.
  - EXAMPLE:

- The following is a reasonably well-structuredprogram.

Each link represents a single link and consequently is given a weight of "1" to start. Let's say the outer loop will be taken exactly four times and inner Loop Can be taken zero or three times Its path expression, with a little work, is:
Path expression: a(b+c)d{e(fi)*fgj(m+l)k}*e(fi)*fgh

- **A:** The flow graph should be annotated by replacing the link name with the maximum of paths through that link (1) and also note the number of times forlooping.
- **B:** Combine the first pair of parallel loops outside the loop and also the pair in the outerloop.
- **C:** Multiply the things out and remove nodes to clear theclutter.

1. For the InnerLoop:
   D:Calculate the total weight of inner loop, which can execute a min. of 0 times and max. of 3 times. So, it inner loop can be evaluated as follows:

   $1^{\underline{3}} = 1^0 + 1^1 + 1^2 + 1^3 = 1 + 1 + 1 + 1 = 4$
2. **E:** Multiply the link weights inside the loop: 1 X 4 =4
3. **F:** Evaluate the loop by multiplying the link wieghts: 2 X 4 =8.
4. **G:** Simpifying the loop further results in the total maximum number of paths in theflowgraph:

   $2 \text{ X } 8^4 \text{ X } 2 = 32{,}768.$

Alternatively, you could have substituted a "1" for each link in the path expression and then simplified, as follows:

a(b+c)d{e(fi)*fgj(m+l)k}*e(fi)*fgh

$= 1(1 + 1)1(1(1 \times 1)^3 1 \times 1 \times 1(1 + 1)1)^4 1(1 \times 1)^3 1 \times 1 \times 1$

$= 2(1^3 1 \times (2))^4 1^3$

$= 2(4 \times 2)^4 \times 4$

$= 2 \times 8^4 \times 4 = 32{,}768$

This is the same result we got graphically.Actually, the outer loop should be taken exactly four times. That doesn't mean it will be taken zero or four times. Consequently, there is a superfluous "4" on the outlink in the last step. Therefore the maximum number of different paths is 8192 rather than 32,768.

STRUCTURED FLOWGRAPH:

Structured code can be defined in several different ways that do not involve ad-hoc rules such as not using GOTOs.

A structured flowgraph is one that can be reduced to a single link by successive application of the transformations of Figure 5.7.

Figure 5.7: Structured Flowgraph Transformations.

The node-by-node reduction procedure can also be used as a test for structured code.Flow graphs that DO NOT contain one or more of the graphs shown below (Figure 5.8) as subgraphs are structured.

1. Jumping intoloops
2. Jumping out ofloops
3. Branching intodecisions
4. Branching out ofdecisions

Figure 5.8: Un-structured sub-graphs.

LOWER PATH COUNT ARITHMETIC:

A lower bound on the number of paths in a routine can be approximated for structured flow graphs.

The arithmetic is as follows:

Case	Path expression	Weight expression
Parallels	A+B	W_A+W_B
Series	AB	$\max(W_A,W_B)$
Loop	$A^{\underline{n}}$	$1, W_1$

The values of the weights are the number of members in a set of paths.
EXAMPLE:

- Applying the arithmetic to the earlier example gives us the identicalsteps unitl step 3 (C) asbelow:

- From Step 4, the it would be different from the previousexample:

- If you observe the original graph, it takes at least two paths to cover and that it can be done in twopaths.
- If you have fewer paths in your test plan than this minimum you probably haven't covered. It's anothercheck.

CALCULATING THE PROBABILITY:

Path selection should be biased toward the low - rather than the high-probability paths.This raises an interesting question:

What is the probability of being at a certain point in a routine?

This question can be answered under suitable assumptions primarily that all probabilities involved are independent, which is to say that all decisions are independent and uncorrelated. We use the same algorithm as before: node-by-node removal of uninteresting nodes.

Weights, Notations and Arithmetic:

- Probabilitiescancomeintotheactonlyatdecisions(includingdecisions associated withloops).
- Annotate each outlink with a weight equal to the probability of going in thatdirection.
- Evidently, the sum of the outlink probabilities must equal1
- For a simple loop, if the loop will be taken a mean of N times, the looping probability is N/(N + 1) and the probability of not looping is 1/(N +1).
- A link that is not part of a decision node has a probability of 1.
- The arithmetic rules are those of ordinaryarithmetic.

Case	Path expression	Weight expression
Parallel	A+B	P_A+P_B
Series	AB	P_AP_B
Loop	A*	$P_A / (1-P_L)$

- In this table, in case of a loop, PA is the probability of the link leaving the loop and PL is the probability oflooping.
- The rules are those of ordinaryprobabilitytheory.
    1. If you can do something either from column A with a probability of PA or from column B with a probability PB, then the probability that you do either is PA +PB.
    2. For the series case, if you must do both things, and their probabilities are independent (as assumed), then the probability that you do both is the product of theirprobabilities.
- For example, a loop node has a looping probability of PL and a probability of not looping of PA, which is obviously equal to I -PL.

$$P_A = 1 - P_L$$

$$P_{NEW} = \frac{P_A}{1-P_L} = \frac{1-P_L}{1-P_L} = 1$$

- Following the above rule, all we've done is replace the outgoing probability with 1 - so why the complicated rule? After a few steps in which you've removed nodes, combined parallel terms, removedloopsand the like, you might find something likethis:

because PL + PA + PB + PC = 1, 1 - PL = PA + PB + PC, and

$$\frac{P_A}{1-P_L} + \frac{P_B}{1-P_L} + \frac{P_C}{1-P_L} = \frac{P_A + P_B + P_C}{1-P_L} = 1$$

which is what we've postulated for any decision. In other words, division by 1 - P_L renormalizes the outlink probabilities so that their sum equals unity after the loop is removed.

EXAMPLE:

- Here is a complicated bit of logic. We want to know the probability associated with cases A, B, andC.

- Let us do this in three parts, starting with case A. Note that the sum of the probabilities at each decision node is equal to 1. Start by throwing away anything that isn't on the way to case A, and then apply the reduction procedure. To avoid clutter, we usually leave out probabilities equal to1.

CASE A:

- Case B issimpler:

- Case C is similar and should yield a probability of 1 - 0.125 - 0.158 = 0.717:

- These checks. It's a good idea when doing this sort of thing to calculate all the probabilities and to verify that the sum of the routine's exit probabilities does equal1.
- If it doesn't, then you've made calculation error or, more likely, you've left out some bra How about path probabilities? That's easy. Just trace the path of interest and multiply the probabilities as yougo.
- Alternatively, write down the path name and do the indicated arithmetic operation.

- Say that a path consisted of links a, b, c, d, e, and the associated probabilities were .2, .5, 1., .01, and I respectively. Path *abcbcbcdeabddea* would have a probability of 5×10^{-10}.
- Long paths are usuallyimprobable.

MEAN PROCESSING TIME OF A ROUTINE:

Given the execution time of all statements or instructions for every link in a flowgraph and the probability for each direction for all decisions are to find the mean processing time for the routine as a whole.

The model has two weights associated with every link: the processing time for that link, denoted by **T**, and the probability of that link **P**.

The arithmetic rules for calculating the mean time:

Case	Path expression	Weight expression
Parallel	A+B	$T_{A+B}=(P_A T_A+P_B T_B)/(P_A+P_B)$ $P_{A+B}= P_A+P_B$
Series	AB	$T_{AB}=T_A+T_B$ $P_{AB} =P_A P_B$
Loop	$A^{\underline{n}}$	$T_A= T_A+ T_L P_L/(1-P_L)$ $P_A =P_A/(1-P_L)$

EXAMPLE:

1. Start with the original flow graph annotated with probabilities and processingtime.

2. Combine the parallel links of the outer loop. The result is just the mean of the processing times for the links because there aren't any other links leaving the first node. Also combine the pair of links at the beginning of the flowgraph.

3. Combine as many serial links as youcan.

4. Use the cross-term step to eliminate a node and to create the inner self - loop.
5.Finally, you can get the mean processing time, by using the arithmetic rulesasfollows:

PUSH/POP, GET/RETURN:
This model can be used to answer several different questions that can turn up in debugging.
It can also help decide which test cases to design.
The question is:

Given a pair of complementary operations such as PUSH (the stack) and POP (the stack), considering the set of all possible paths through the routine, what is the net effect of the routine? PUSH or POP? How many times? Under what conditions?

Here are some other examples of complementary operations to which this model applies:

GET/RETURN a resource block.

OPEN/CLOSE a file.

START/STOP a device or process.

EXAMPLE 1 (PUSH / POP):

- Here is the Push/PopArithmetic:

Case	Path expression	Weight expression
Parallels	A+B	W_A+W_B
Series	AB	W_AW_B
Loop	A^*	W_A^*

- The numeral 1 is used to indicate that nothing of interest (neither PUSH nor POP) occurs on a givenlink.
- "H" denotes PUSH and "P" denotes POP. The operationsare commutative, associative, anddistributive.

PUSH/POP MULTIPLICATION TABLE

×	H PUSH	P POP	1 NONE
H	H^2	1	H
P	1	P^2	P
1	H	P	1

PUSH/POP ADDITION TABLE

+	H PUSH	P POP	1 NONE
H	H	P + H	H + 1
P	P + H	P	P + 1
1	H + 1	P + 1	1

- Consider the following flowgraph:

$P(P + 1)1\{P(HH)^{n1}HP1(P + H)1\}^{n2}P(HH)^{n1}HPH$

- Simplifying by using the arithmetictables,
  $= (P^2 + P)\{P(HH)^{n1}(P + H)\}^{n1}(HH)^{n1}$
  $= (P^2 + P)\{H^{2n1}(P^2 + 1)\}^{n2}H^{2n1}$
- Below Table 5.9 shows several combinations of values for the twolooping terms - M1 is the number of times the inner loop will be taken and M2 the number of times the outer loop will be taken.

M_1	M_2	PUSH/POP
0	0	$P + P^2$
0	1	$P + P^2 + P^3 + P^4$
0	2	$\sum_{1}^{6} P^i$
0	3	$\sum_{1}^{8} P^i$
1	0	$1 + H$
1	1	$\sum_{0}^{3} H^i$
1	2	$\sum_{0}^{5} H^i$
1	3	$\sum_{0}^{7} H^i$
2	0	$H^2 + H^3$
2	1	$\sum_{4}^{7} H^i$
2	2	$\sum_{6}^{11} H^i$
2	3	$\sum_{8}^{15} H^i$

Figure 5.9: Result of the PUSH / POP Graph Analysis.

- These expressions state that the stack will be popped only if the inner loop is nottaken.
- The stack will be left alone only if the inner loop is iterated once, but it may also bepushed.
- For all other values of the inner loop, the stack will only bepushed.

EXAMPLE 2 (GET / RETURN):

- Exactlythesamearithmetictablesusedforpreviousexampleareusedfor GET / RETURN a buffer block or resource, or, in fact, for any pairof

complementary operations in which the total number of operations in either direction iscumulative.

- The arithmetic tables for GET/RETURNare:

Multiplication Table

×	G	R	1
G	G^2	1	G
R	1	R^2	R
1	G	R	1

Addition Table

+	G	R	1
G	G	G + R	G + 1
R	G + R	R	R + 1
1	G + 1	R + 1	1

"G" denotes GET and "R" denotes RETURN.

- Consider the followingflowgraph:

- G(G +R)G(GR)*GGR*R
  = G(G + R)G^3R*R
  = (G +R)G^3R*
  = (G^4 +G^2)R*
- This expression specifies the conditions under which the resources will be balanced on leaving theroutine.
- If the upper branch is taken at the first decision, the second loop must be taken fourtimes.
- If the lower branch is taken at the first decision, the second loop must be takentwice.
- For any other values, the routine will not balance. Therefore, the first loop does not have to be instrumented to verify this behavior because its impact should benil.

LIMITATIONS AND SOLUTIONS:

- The main limitation to these applications is the problem of unachievablepaths.
- The node-by-node reduction procedure, and most graph-theory-based algorithms work wellwhenallpathsarepossible,butmayprovidemisleadingresultswhensome pathsare unachievable.
- The approach to handling unachievable paths (for any application) is to partitionthe graph into subgraphs so that all paths in each of the subgraphs areachievable.
- The resulting subgraphs may overlap, because one path may be common to several differentsubgraphs.
- Eachpredicate'struth-functionalvaluepotentiallysplitsthegraphintotwosubgraphs. For n predicates, there could be as many as 2^nsubgraphs.

REGULAR EXPRESSIONS AND FLOW ANOMALY DETECTION:

- **THEPROBLEM:**
  - The generic flow-anomaly detection problem (note: not just data-flowanomalies, but any flow anomaly) is that of looking for a specific sequence of options considering all possible paths through aroutine.
  - Let the operations be SET and RESET, denoted by s and r respectively, and we want to know if there is a SET followed immediately a SET or a RESET followed immediately by a RESET (an *ss* or an *rr*sequence).
  - Some more applicationexamples:
    1. A file can be opened (o), closed (c), read (r), or written (w). If the file is read or written to after it's been closed, the sequence is nonsensical. Therefore, *cr* and *cw* are anomalous. Similarly, if the file is read before it's been written, just after opening, we may have a bug. Therefore, *or* is also anomalous. Furthermore, *oo* and *cc*, though not actual bugs, are a waste of time and therefore should also beexamined.
    2. A tape transport can do a rewind (d), fast-forward (f), read (r), write (w), stop (p), and skip (k). There are rules concerning the use of the transport; for example, you cannot go from rewind to fast-forward without an intervening stop or from rewind or fast-forward to read or write withoutan intervening stop. The following sequences are anomalous: *df*, *dr*, *dw*, *fd*, and *fr*. Does the flowgraph lead to anomalous sequences on any path? If so, what sequences and under whatcircumstances?
    3. The data-flow anomalies discussed in Unit 4 requires us to detect the *dd*, *dk*, *kk*, and *ku* sequences. Are there paths with anomalous dataflows?

- THEMETHOD:
  - Annotate each link in the graph with the appropriate operator or the null operator 1.
  - Simplify things to the extent possible, using the fact that a + a = a and 12 =1.
  - You now have a regular expression that denotes all the possible sequences of operators in that graph. You can now examine that regular expression for the sequences ofinterest.
  - **EXAMPLE:** Let A, B, C, be nonempty sets of character sequences whose smallest string is at least one character long. Let T be a two-character string of characters. Then if T is a substring of (i.e., if T appears within) AB^nC, then T will appear in AB^2C. (**HUANG'sTheorem**)
    As an example, let
  - A = *pp*
    B = *srr*
    C = *rp*
    T =*ss*

    The theorem states that *ss* will appear in $pp(srr)^n rp$ if it appears in $pp(srr)^2 rp$.

- However,let

  $A = p + pp + ps$
  $B = psr + ps(r + ps)$
  $C = rp$
  $T = P^4$

  Is it obvious that there is a p^4 sequence in AB^nC? The theorem states that we have only to look at

  $(p + pp + ps)[psr + ps(r + ps)]^2 rp$

  Multiplying out the expression and simplifying shows that there is no p^4 sequence.
- Incidentally, the above observation is an informal proof of the wisdom of looping twice discussed in Unit 2. Because data-flow anomalies are represented by two-character sequences, it follows the above theorem that looping twice is what you need to do to find suchanomalies.

- LIMITATIONS:
  - Huang's theorem can be easily generalized to cover sequences of greater length than two characters. Beyond three characters, though, things get complex and this method has probably reached its utilitarian limit for manualapplication.
  - There are some nice theorems for finding sequences that occur at the beginnings and ends of strings but no nice algorithms for finding strings buried in an expression.
  - Static flow analysis methods can't determine whether a path is or is not achievable. Unless the flow analysis includes symbolic execution or similar techniques, the impact of unachievable paths will not be included in theanalysis.

  The flow-anomaly application, for example, doesn't tell us that there will be a flow anomaly - it tells us that if the path is achievable, then there will be a flow anomaly. Such analytical problems go away, of course, if you take the trouble to design routines for which all paths are achievable.

LOGIC BASED TESTING

OVERVIEW OF LOGIC BASED TESTING:

- **INTRODUCTION:**
  - The functional requirements of many programs can be specified by **decision tables**, which provide a useful basis for program and testdesign.
  - Consistency and completeness can be analyzed by using boolean algebra, which can also be used as a basis for test design. Boolean algebra is trivialized by using **Karnaugh-Veitchcharts**.
  - "Logic" is one of the most often used words in programmers' vocabularies but one of their least usedtechniques.
  - Boolean algebra is to logic as arithmetic is to mathematics. Without it, the tester or programmer is cut off from many test and design techniques and tools that incorporate thosetechniques.
  - Logic has been, for several decades, the primary tool of hardwarelogicdesigners.
  - Many test methods developed for hardware logic can be adapted to software logic testing. Because hardware testing automation is 10 to 15 years ahead of software testing automation, hardware testing methods and its associated theory is a fertile ground for software testingmethods.
  - As programming and test techniques have improved, the bugs have shifted closer to the process front end, to requirements and their specifications. These bugs range from 8% to 30% of the total and because they're first-in and last-out, they're the costliest ofall.
  - The trouble with specifications is that they're hard toexpress.
  - Boolean algebra (also known as the sentential calculus) is the most basic of all logic systems.
  - Higher-order logic systems are needed and used for formalspecifications.
  - Much of logical analysis can be and is embedded in tools. But these tools incorporate methods to simplify, transform, and check specifications, and the methods are to a large extent based on booleanalgebra.

  - KNOWLEDGE BASEDSYSTEM:

    - The **knowledge-based system** (also expert system, or "artificial intelligence" system) has become the programming construct of choice for many applications that were once considered verydifficult.
    - Knowledge-based systems incorporate knowledge from a knowledge domain such as medicine, law, or civil engineering into a database. The data can then be queried and interacted with to provide solutions to problems in thatdomain.
    - One implementation of knowledge-based systems is to incorporate the expert's knowledge into a set of rules. The user can then provide data and ask questions based on thatdata.

- The user's data is processed through the rule base to yield conclusions (tentative or definite) and requests for more data. The processing is done by a program called the **inferenceengine**.
- Understanding knowledge-based systems and their validation problems requires an understanding of formallogic.

- Decision tables are extensively used in business data processing; Decision-table preprocessors as extensions to COBOL are in common use; boolean algebra is embedded in the implementation of theseprocessors.
- Although programmed tools are nice to have, most of the benefits of boolean algebra can be reaped by wholly manual means if you have the right conceptual tool: the Karnaugh-Veitch diagram is that conceptualtool.

- DECISIONTABLES:

- Figure 6.1 is a limited - entry decision table. It consists of four areas called the condition stub, the condition entry, the action stub, and the actionentry.
- Eachcolumn of thetableisarulethatspecifiestheconditionsunder whichtheactions named in the action stub will takeplace.
- The condition stub is a list of names of conditions.

CONDITION ENTRY

		RULE 1	RULE 2	RULE 3	RULE 4
CONDITION STUB	CONDITION 1	YES	YES	NO	NO
	CONDITION 2	YES	I	NO	I
	CONDITION 3	NO	YES	NO	I
	CONDITION 4	NO	YES	NO	YES
ACTION STUB	ACTION 1	YES	YES	NO	NO
	ACTION 2	NO	NO	YES	NO
	ACTION 3	NO	NO	NO	YES

ACTION ENTRY

Figure 6.1 : Examples of Decision Table.

- A more general decision table can be asbelow:

Printer troubleshooter

		Rules							
Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognised	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

Figure 6.2 : Another Examples of Decision Table.

- A rule specifies whether a condition should or should not be met for the rule to be satisfied. "YES" means that the condition must be met, "NO" means that the condition must not be met, and "I" means that the condition plays no part in the rule, or it is immaterial to thatrule.

The action stub names the actions the routine will take or initiate if the rule is satisfied.

- If the action entry is "YES", the action will take place; if "NO", the action will not take place.

The table in Figure 6.1 can be translated as follows:

Action 1 will take place if conditions 1 and 2 are met and if conditions 3 and 4 are not met (rule 1) or if conditions 1, 3, and 4 are met (rule2).

- "Condition" is another word forpredicate.
- Decision-tableuses"condition"and"satisfied"or"met". Letususe"predicate"and TRUE /FALSE.
- Now the above translationsbecome:
    1. Action 1 will be taken if predicates 1 and 2 are true and if predicates 3 and 4 are false (rule 1), or if predicates 1, 3, and 4 are true (rule2).
    2. Action 2 will be taken if the predicates are all false, (rule3).
    3. Action 3 will take place if predicate 1 is false and predicate 4 is true (rule4).
- In addition to the stated rules, we also need a **Default Rule** that specifies the defaultaction to be taken when all other rules fail. The default rules for Table in Figure 6.1 is shown in Figure6.3

	Rule 5	Rule 6	Rule 7	Rule 8
CONDITION 1	I	NO	YES	YES
CONDITION 2	I	YES	I	NO
CONDITION 3	YES	I	NO	NO
CONDITION 4	NO	NO	YES	I
DEFAULT ACTION	YES	YES	YES	YES

Figure 6.3 : The default rules of Table in Figure 6.1

- **DECISION-TABLEPROCESSORS:**
    - Decision tables can be automatically translated into code and, as such, are a higher-orderlanguage
    - If the rule is satisfied, the corresponding action takesplace
    - Otherwise, rule 2 is tried. This process continues until either a satisfiedrule results in an action or no rule is satisfied and the default action istaken
    - Decision tables have become a useful tool in the programmers kit, inbusiness data processing.

DECISION-TABLES AS BASIS FOR TEST CASE DESIGN:

1. The specification is given as a decision table or can be easily converted intoone.
2. The order in which the predicates are evaluated does not affect interpretation of the rules or the resulting action - i.e., an arbitrary permutation of the predicate order will not, or should not, affect which action takesplace.
3. The order in which the rules are evaluated does not affect the resulting action - i.e., an arbitrary permutation of rules will not, or should not, affect which action takes place.
4. Once a rule is satisfied and an action selected, no other rule need beexamined.
5. If several actions can result from satisfying a rule, the order in which the actions are executed doesn'tmatter.

DECISION-TABLES AND STRUCTURE:

- Decision tables can also be used to examine a program'sstructure.
- Figure 6.4 shows a program segment that consists of a decisiontree.
- These decisions, in various combinations, can lead to actions 1, 2, or3.

Figure 6.4 : A Sample Program

- If the decision appears on a path, put in a YES or NO as appropriate. If the decision does not appear on the path, put in an I, Rule 1 does not contain decision C, therefore its entries are: YES, YES, I,YES.
- The corresponding decision table is shown in Table6.1

	RULE 1	RULE 2	RULE 3	RULE 4	RULE 5	RULE 6
CONDITION A **CONDITION B** **CONDITION C** **CONDITION D**	YES YES I YES	YES NO I I	YES YES I NO	NO I YES I	NO I NO YES	NO I NO NO
ACTION 1 **ACTION 2** **ACTION 3**	YES NO NO	YES NO NO	NO YES NO	NO YES NO	NO YES NO	NO NO YES

Table 6.1: Decision Table corresponding to Figure 6.4

As an example, expanding the immaterial cases results as below:

	RULE 1	RULE 2
CONDITION 1	YES	YES
CONDITION 2	I	NO
CONDITION 3	YES	I
CONDITION 4	NO	NO
ACTION 1	YES	NO
ACTION 2	NO	YES

RULE 1.1	RULE 1.2	RULE 2.1	RULE 2.2
YES	YES	YES	YES
YES	NO	NO	NO
YES	YES	YES	NO
NO	NO	NO	NO
YES	YES	NO	NO
NO	NO	YES	YES

Table 6.2: Expansion of Table 6.1

- Similalrly, If we expand the immaterial cases for the above Table 6.1, it results in Table 6.2 asbelow:

	R 1	RULE 2	R 3	RULE 4	R 5	R 6
CONDITION A	YY	YYYY	YY	NNNN	NN	NN
CONDITION B	YY	NNNN	YY	YYNN	NY	YN
CONDITION C	YN	NNYY	YN	YYYY	NN	NN
CONDITION D	YY	YNNY	NN	NYYN	YY	NN

1. Sixteen cases are represented in Table 6.1, and no case appearstwice.
2. Consequently, the flowgraph appears to be complete andconsistent.
3. As a first check, before you look for all sixteen combinations, count the number of Y's and N's in each row. They should be equal. We can find the bug that way.

- ANOTHER EXAMPLE - A TROUBLE SOMEPROGRAM:

1. Consider the following specification whose putative flowgraph is shown inFigure 6.5:
    1. If condition A is met, do process A1 no matter what other actions are taken or what other conditions aremet.

2. If condition B is met, do process A2 no matter what other actions are taken or what other conditions aremet.
3. If condition C is met, do process A3 no matter what other actions are taken or what other conditions aremet.
4. If none of the conditions is met, then do processes A1, A2, andA3.
5. When more than one process is done, process A1 must be done first, then A2, and then A3. The only permissible cases are: (A1), (A2), (A3), (A1,A3), (A2,A3) and(A1,A2,A3).

2. Figure 6.5 shows a sample program with abug.

Figure 6.5 : A Troublesome Program

- The programmer tried to force all three processes to be executed for the $\bar{A}\bar{B}\bar{C}$cases but forgot that the B and C predicateswould be done again, thereby bypassing processes A2 and A3.
- Table 6.3 shows the conversion of this flow graph into a decision table after expansion.

RULES

	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}C$	$\bar{A}BC$	$\bar{A}B\bar{C}$	$AB\bar{C}$	ABC	$A\bar{B}C$	$A\bar{B}\bar{C}$
CONDITION A	NO	NO	NO	NO	YES	YES	YES	YES
CONDITION B	NO	NO	YES	YES	YES	YES	NO	NO
CONDITION C	NO	YES	YES	NO	NO	YES	YES	NO
ACTION 1	YES	NO	NO	NO	YES	YES	YES	YES
ACTION 2	YES	NO	YES	YES	YES	YES	NO	NO
ACTION 3	YES	YES	YES	NO	NO	YES	YES	NO

Table 6.3: Decision Table for Figure 6.5

PATH EXPRESSIONS:

- **GENERAL:**
  - Logic-based testing is structural testing when it's applied to structure (e.g., control flow graph of an implementation); it's functional testing when it's applied to a specification.
  - In logic-based testing we focus on the truth values of control flowpredicates.

- A **predicate** is implemented as a process whose outcome is a truth-functional value.
- For our purpose, logic-based testing is restricted to binarypredicates.
- We start by generating path expressions by path tracing as in Unit V, but this time, our purpose is to convert the path expressions into boolean algebra, using the predicates' truth values (e.g., A and $\bar{A}$) asweights.

- BOOLEAN ALGEBRA:
  - **STEPS:**
    1. Label each decision with an uppercase letter that represents the truth value of the predicate. The YES or TRUE branch is labeled with a letter (say A)and theNOorFALSEbranchwiththesameletteroverscored(say$\bar{A}$).
    2. The truth value of a path is the product of the individual labels. Concatenation or products mean "AND". For example, the straight-through path of Figure 6.5, which goes via nodes 3, 6, 7, 8, 10, 11, 12, and 2, has a truth value of ABC. The path via nodes 3, 6, 7, 9 and 2 has a value of$AB\bar{C}$..
    3. If two or more paths merge at a node, the fact is expressed by use of a plus sign (+) which means"OR".

Figure 6.5: A Troublesome Program

  - Using this convention, the truth-functional values for several of the nodes can be expressed in terms of segments from previous nodes. Use the node name to identify thepoint.

$$N6 = A + \bar{A}\bar{B}\bar{C}$$
$$N8 = (N6)B + \bar{A}B = AB + \bar{A}\bar{B}\bar{C}B + \bar{A}B$$
$$N11 = (N8)C + (N6)\bar{B}C$$
$$N12 = N11 + \bar{A}\bar{B}C$$
$$N2 = N12 + (N8)\bar{C} + (N6)\bar{B}\bar{C}$$

  - There are only two numbers in boolean algebra: zero (0) and one (1). One means"always true" and zero means "alwaysfalse".

- RULES OF BOOLEANALGEBRA:
  - Booleanalgebrahasthreeoperators:X(AND),+(OR) and $\overline{A}$(NOT)
  - **X :** meaning AND. Also called multiplication. A statement such as AB (A X B) means "A and B are both true". This symbol is usually left out as in ordinaryalgebra.
  - **+ :** meaning OR. "A + B" means "either A is true or B is true or both".
  - $\overline{A}$meaning NOT. Also negation or complementation. This is read as either "not A" or "A bar". The entire expression under the bar isnegated.
  - The following are the laws of booleanalgebra:

1. $A + A$	$= A$	If something is true, saying it
$\overline{A} + \overline{A}$	$= \overline{A}$	twice doesn't make it truer, ditto for falsehoods.
2. $A + 1$	$= 1$	If something is always true, then "either A or true or both" must also be universally true.
3. $A + 0$	$= A$	
4. $A + B$	$= B + A$	Commutative law.
5. $A + \overline{A}$	$= 1$	If either A is true or not-A is true, then the statement is always true.
6. AA	$= A$	
$\overline{A}\overline{A}$	$= \overline{A}$	
7. $A \times 1$	$= A$	
8. $A \times 0$	$= 0$	
9. AB	$= BA$	
10. $A\overline{A}$	$= 0$	A statement can't be simultaneously true and false.
11. $\overline{\overline{A}}$	$= A$	"You ain't not going" means you are. How about, "I ain't not never going to get this nohow."?
12. $\overline{0}$	$= 1$	
13. $\overline{1}$	$= 0$	
14. $\overline{A + B}$	$= \overline{A}\,\overline{B}$	Called "De Morgan's theorem or law."
15. $\overline{AB}$	$= \overline{A} + \overline{B}$	
16. $A(B + C)$	$= AB + AC$	Distributive law.
17. $(AB)C$	$= A(BC)$	Multiplication is associative.
18. $(A + B) + C$	$= A + (B + C)$	So is addition.
19. $A + \overline{A}B$	$= A + B$	Absorptive law.
20. $A + AB$	$= A$	

In all of the above, a letter can represent a single sentence or an entire boolean algebra expression.
Individual letters in a boolean algebra expression are called **Literals** (e.g. A,B)
The product of several literals is called a **product term** (e.g., ABC, DE).
An arbitrary boolean expression that has been multiplied out so that it consists of the sum of products (e.g., ABC + DEF + GH) is said to be in **sum-of-products form**.
The result of simplifications (using the rules above) is again in the sum of product form and each product term in such a simplified version is called a **prime implicant**. For example, ABC + AB + DEF reduce by rule 20 to AB + DEF; that is, AB and DEF are prime implicants.
The path expressions of Figure 6.5 can now be simplified by applying the rules.
The following are the laws of boolean algebra:

$$
\begin{aligned}
N6 &= A + \overline{A}\,\overline{B}\,\overline{C} \\
&= A + \overline{B}\,\overline{C} && \text{: Use rule 19, with "B" = } \overline{B}\,\overline{C}. \\
N8 &= (N6)B + \overline{A}B \\
&= (A + \overline{B}\,\overline{C})B + \overline{A}B && \text{: Substitution.} \\
&= AB + \overline{B}\,\overline{C}B + \overline{A}B && \text{: Rule 16 (distributive law).} \\
&= AB + B\overline{B}C + \overline{A}B && \text{: Rule 9 (commutative multiplication).} \\
&= AB + 0C + \overline{A}B && \text{: Rule 10.} \\
&= AB + 0 + \overline{A}B && \text{: Rule 8.} \\
&= AB + \overline{A}B && \text{: Rule 3.} \\
&= (A + \overline{A})B && \text{: Rule 16 (distributive law).} \\
&= 1 \times B && \text{: Rule 5.} \\
&= B && \text{: Rules 7, 9.}
\end{aligned}
$$

Similarly,

$$
\begin{aligned}
N11 &= (N8)C + (N6)\overline{B}C \\
&= BC + (A + \overline{B}\,\overline{C})\overline{B}C && \text{: Substitution.} \\
&= BC + A\overline{B}C && \text{: Rules 16, 9, 10, 8, 3.} \\
&= C(B + \overline{B}A) && \text{: Rules 9, 16.} \\
&= C(B + A) && \text{: Rule 19.} \\
&= AC + BC && \text{: Rules 16, 9, 9, 4.} \\
N12 &= N11 + \overline{A}\,\overline{B}C \\
&= AC + BC + \overline{A}\,\overline{B}C \\
&= C(B + \overline{A}\,\overline{B}) + AC \\
&= C(\overline{A} + B) + AC \\
&= C\overline{A} + AC + BC \\
&= C + BC \\
&= C \\
N2 &= N12 + (N8)\overline{C} + (N6)\overline{B}\,\overline{C} \\
&= C + B\overline{C} + (A + \overline{B}\,\overline{C})\overline{B}\,\overline{C} \\
&= C + B\overline{C} + \overline{B}\,\overline{C} \\
&= C + \overline{C}(B + \overline{B}) \\
&= C + \overline{C} \\
&= 1
\end{aligned}
$$

The deviation from the specification is now clear. The functions should have been:

$$
\begin{aligned}
N6 &= A + \overline{A}\,\overline{B}\,\overline{C} = A + \overline{B}\,\overline{C} && \text{: correct.} \\
N8 &= B + \overline{A}\,\overline{B}\,\overline{C} = B + \overline{A}\,\overline{C} && \text{: wrong, was just B.} \\
N12 &= C + \overline{A}\,\overline{B}\,\overline{C} = C + \overline{A}\,\overline{B} && \text{: wrong, was just C.}
\end{aligned}
$$

Loops complicate things because we may have to solve a boolean equation to determine what predicate value combinations lead to where.

KV CHARTS:

- **INTRODUCTION:**
  - If you had to deal with expressions in four, five, or six variables, you could get bogged down in the algebra and make as many errors in designing test cases as there are bugs in the routine you'retesting.
  - **Karnaugh-Veitch chart** reduces boolean algebraic manipulations to graphical trivia.
  - Beyond six variables these diagrams get cumbersome and may not beeffective.
- SINGLEVARIABLE:
  - Figure 6.6 shows all the boolean functions of a single variable and their equivalent representation as a KVchart.

Figure 6.6 : KV Charts for Functions of a Single Variable.

- The charts show all possible truth values that the variable A can have.
- A "1" means the variable’s value is "1" or TRUE. A "0" means that the variable's value is 0 orFALSE.
- The entry in the box (0 or 1) specifies whether the function that thechart represents is true or false for that value of thevariable.
- Weusuallydonotexplicitlyputin0entriesbutspecifyonlytheconditionsunder which the function istrue.

- TWOVARIABLES:
  - Figure 6.7 shows eight of the sixteen possible functions of twovariables.

Figure 6.7: KV Charts for Functions of Two Variables.

- Each box corresponds to the combination of values of the variables for the row and column of thatbox.
- A pair may be adjacent either horizontally or vertically but notdiagonally.
- Any variable that changes in either the horizontal or vertical direction does not appear in theexpression.
- In the fifth chart, the B variable changes from 0 to 1 going down the column, and because the A variable's value for the column is 1, the chart is equivalent to a simpleA.
- Figure 6.8 shows the remaining eight functions of twovariables.

Figure 6.8: More Functions of Two Variables.

- The first chart has two 1's in it, but because they are not adjacent, each must be takenseparately.
- They are written using a plussign.
- It is clear now why there are sixteen functions of twovariables.
- Each box in the KV chart corresponds to a combination of the variables'values.
- That combination might or might not be in the function (i.e., the box corresponding to that combination might have a 1 or 0entry).
- Since n variables lead to 2^n combinations of 0 and 1 for the variables, and each such combination (box) can be filled or not filled, leading to 2^{2n} ways of doing this.
- Consequently for one variable there are 2^{21} = 4 functions, 16 functions of 2 variables, 256 functions of 3 variables, 16,384 functions of 4 variables, andsoon.

- Given two charts over the same variables, arranged the same way, their product is the term by term product, their sum is the term by term sum, and the negation of a chart is gotten by reversing all the 0 and 1 entries in thechart.

OR

- THREEVARIABLES:
  - KV charts for three variables are shownbelow.
  - As before, each box represents an elementary term of three variables with a bar appearing or not appearing according to whether the row-column heading forthat box is 0 or1.
  - A three-variable chart can have groupings of 1, 2, 4, and 8boxes.
  - A few examples will illustrate theprinciples:

Figure 6.8: KV Charts for Functions of Three Variables.

- You'llnoticethattherearesseveralwaystocircletheboxesintomaximum-sized coveringgroups.

PART A(2 MARKS)

1. Define Goal intesting
   A Goal can be described as A statement of intent A statement of a accomplishment that an individual

2. What are the three types of goals in
   testing BusinessGoal
   Technical
   Goal Political
   Goal

3. Define the termpolicy
   A policy can be defined as a high-level statement of principle or course of action that is used to govern a set of activities in an organization

4. Define TestPlan
   A Plan is a document that provides a frame work or approach for achieving a set of goals.

5. DefineFeatures
   Features may be described as distinguishing characteristics of a software component or system

6. Define the term Pass / FailCriteria
   Given a test item and a test case, the tester must have a set of criteria to decide on whether the test has been passed or failed upon execution

7. Define TestProcedure
   A Procedure in general as a sequence of steps required to carry out a specific task

8. . Define Test SummaryReport
   This report is prepared when testing is complete. It is summary of the results of the testing efforts. It also becomes a part of the projects historical database and provides a basis for lessons learned as applied to future projects.

9. What is the use of V-model intesting
   The V-model is model that illustrates how testing activities can be integrated in to each phase of the standard software life cycle

10. What are the various approaches to test
    costestimation COCOMOModel
    Use of test cost drivers
    Test Tasks
    Testers / Developers
    ratio Expert judgment

PART B(10 MARKS)

1. Explain the problem occurred in the regular expressions with an example? b.Explain which method will be useful for regular expressions with anexample?
2. Discuss regular expressions and flowanomalydetection?
3. Explain a regular expression and flow anomaly detectionmethod With an exampleandlimitations
4. Demonstrate how to find approximate minimum no. of paths withanExample?
5. Explain the probability of getting path expression withanexample?

UNIT V
STATES, STATE GRAPHS, AND TRANSITION TESTING

Introduction

- Thefinitestatemachineisasfundamentaltosoftwareengineeringasbooleanalgebrato logic.
- State testing strategies are based on the use of finite state machine models for software structure, software behavior, or specifications of softwarebehavior.
- Finite state machines can also be implemented as table-driven software, in whichcase they are a powerful designoption.

State Graphs

- A state is defined as: “A combination of circumstances or attributes belonging forthe time being to a person orthing.”
- For example, a moving automobile whose engine is running can have the followingstates with respect to its transmission.
    - Reverse gear
    - Neutralgear
    - Firstgear
    - Secondgear
    - Thirdgear
    - Fourth gear

State graph -Example

- For example, a program that detects the character sequence “ZCZC” can be inthe followingstates.
- Neither ZCZC nor any part of it has beendetected.
    - Z has beendetected.
    - ZC has beendetected.
    - ZCZ has been detected.
    - ZCZC has beendetected.

States are represented by Nodes. State are numbered or may identified by words or whatever else is convenient.

Inputs and Transitions

- Whatever is being modeled is subjected to inputs. As a result of those inputs, the state changes, or is said to have made aTransition.
- Transitions are denoted by links that jointhestates.
- The input that causes the transition are marked on the link; that is, the inputs are link weights.
- There is one out link from every state for everyinput.

- If several inputs in a state cause a transition to the same subsequent state, instead of drawing a bunch of parallel links we can abbreviate the notation by listing the several inputs as in: “input1, input2,input3……”.

Finite StateMachine

- A finite state machine is an abstract device that can be represented by a state graph having a finite number of states and a finite number of transitions betweenstates.
  - oOutputs
- An output can be associated with anylink.
- Out puts are denoted by letters or words and are separated from inputs by a slashas follows:“input/output”.
- As always, output denotes anything of interest that’s observable and is not restricted to explicit outputs bydevices.
- Outputs are also linkweights.
- If every input associated with a transition causes the same output, then denoted itas:
  - “input1,input2,input3…..... /output”

State Tables

- Big state graphs are cluttered and hard tofollow.
- It’s more convenient to represent the state graph as a table (the state table or state transition table) that specifies the states, the inputs, the transitions and the outputs.
- The following conventions areused:
- Each row of the table corresponds to astate.
- Each column corresponds to an inputcondition.
- The box at the intersection of a row and a column specifies the next state (the transition) and the output, ifany.

State Table-Example

inputs

STATE	Z	C	A
NONE	Z	NONE	NONE
Z	Z	ZC	NONE
ZC	ZCZ	NONE	NONE
ZCZ	Z	ZCZC	NONE
ZCZC	ZCZC	ZCZC	ZCZC

Time Versus Sequence

- State graphs don't represent time-they represent sequence.
- A transition might take microseconds orcenturies;
- A system could be in one state for milliseconds and another for years- the stategraph would be the same because it has no notion oftime.
- Although the finite state machines model can be elaborated to include notions of time in addition to sequence, such as time PetriNets.
    - oSoftware implementation
- There is rarely a direct correspondence between programs and the behavior of aprocess described as a stategraph.
- Thestategraphrepresents,the totalbehaviorconsistingofthetransport,thesoftware,the executive, the status returns, interrupts, and soon.
- There is no simple correspondence between lines of code and states. The state table forms thebasis.

Good State Graphs and Bad

- What constitutes a good or a bad state graph is to some extent biased by the kinds of state graphs that are likely to be used in a software test designcontext.
- Here are some principles forjudging.
    - The total number of states is equal to the product of the possibilities of factorsthat make up thestate.
    - For every state and input there is exactly one transition specified to exactly one, possibly the same,state.
    - For every transition there is one output action specified. The output could be trivial, but at least one output does somethingsensible.
    - For every state there is a sequence of inputs that will drive the system back to the same state.

Important graphs

State Bugs-Number of States

- The number of states in a state graph is the number of states we choose to recognize or model.

- The state is directly or indirectly recorded as a combination of values of variables that appear in the database.
- For example, the state could be composed of the value of a counter whose possible values ranged from 0 to 9, combined with the setting of two bit flags, leading to a total of 2*2*10=40states.
- The number of states can be computed asfollows:
    - Identify all the component factors of thestate.
    - Identify all the allowable values for eachfactor.
    - The number of states is the product of the number of allowable values of all the factors.
- Before you do anything else, before you consider one test case, discuss the number of states you think there are with the number of states the programmer thinks thereare.
- There is no point in designing tests intended to check the system's behavior in various states if there's no agreement on how many states thereare.

oImpossible States

- Some times some combinations of factors may appear tobeimpossible.
- The discrepancy between the programmer's state count and the tester's state count is often due to a difference of opinion concerning "impossiblestates".
- A robust piece of software will not ignore impossible states but will recognize them and invoke an illogical condition handler when they appear to have occurred.

Equivalent States

- Two states are Equivalent if every sequence of inputs starting from one state produces exactly the same sequence of outputs when started from the other state. This notion can also be extended to set ofstates.

Merging of Equivalent States

Recognizing Equivalent States

- Equivalent states can be recognized by the followingprocedures:
- Therowscorresponding tothetwostatesareidenticalwithrespecttoinput/output/next state but the name of the next state coulddiffer.
- There are two sets of rows which, except for the state names, have identical state graphs with respect to transitions and outputs. The two sets can bemerged.

TransitionBugs-
unspecified and contradictory Transitions

- Every input-state combination must have a specifiedtransition.
- If the transition is impossible, then there must be a mechanism that prevents the input from occurring in thatstate.
- Exactly one transition must be specified for every combination of input andstate.
- A program can't have contradictions orambiguities.
- Ambiguitiesareimpossiblebecausetheprogramwilldosomethingfor everyinput.Even the state does not change, by definition this is a transition to the samestate.

Unreachable States

- An unreachable state is like unreachablecode.
- A state that no input sequence canreach.
- An unreachable state is not impossible, just as unreachable code is notimpossible
- There may be transitions from unreachable state to other states; there usually becausethe state became unreachable as a result of incorrecttransition.
- There are two possibilities for unreachablestates:
    - There is a bug; that is some transitions aremissing.
    - The transitions are there, but you don't know aboutit.

Dead States

- A dead state is a state that once entered cannot beleft.
- This is not necessarily a bug but it issuspicious.

Output Errors

- The states, transitions, and the inputs could be correct, there could be no dead or unreachable states, but the output for the transition could beincorrect.
- Output actions must be verified independently of states andtransitions.

StateTesting

Impact of Bugs

- If a routine is specified as a state graph that has been verified as correct in alldetails. Program code or table or a combination of both must still beimplemented.
- A bug can manifest itself as one of the followingsymptoms:
- Wrong number ofstates.
- Wrong transitions for a given state-inputcombination.
- Wrong output for a giventransition.
- Pairsofstatesorsetsofstatesthatareinadvertentlymadeequivalent.
- States or set of states that are split to create in equivalentduplicates.

- States or sets of states that have becomedead.
- States or sets of states that have becomeunreachable.

Principles of State Testing

- The strategy for state testing is analogous to that used for path testing flowgraphs.
- Just as it's impractical to go through every possible path in a flow graph, it's impractical to go through every path in a stategraph.
- The notion of coverage is identical to that used for flowgraphs.
- Even though more state testing is done as a single case in a grand tour, it's impractical to do it that way for severalreasons.
- In the early phases of testing, you will never complete the grand tour because ofbugs.
- Later, in maintenance, testing objectives are understood, and only a few of the states and transitions have to be tested. A grand tour is waste oftime.
- Theirs is no much history in a long test sequence and so much has happened that verification isdifficult.

Starting point of state testing

- Define a set of covering input sequences that get back to the initial state whenstarting from the initialstate.
- For each step in each input sequence, define the expected next state, the expected transition, and the expected outputcode.
- A set of tests, then, consists of three sets ofsequences:
    - Inputsequences
    - Corresponding transitions or next-statenames
    - Outputsequences

Limitations and Extensions

- State transition coverage in a state graph model does not guarantee completetesting.
- How defines a hierarchy of paths and methods for combining paths to produce coversof state graphs.
- The simplest is called a "0 switch" which corresponds to testing eachtransition individually.
- The next level consists of testing transitions sequences consisting of twotransitions called "1switches".
- The maximum length switch is "n-1 switch" where there are n numbers ofstates.
    - Situations at which state testing is useful
- Any processing where the output is based on the occurrence of one or more sequences of events, such as detection of specified input sequences, sequential format validation, parsing, and other situations in which the order of inputs isimportant.
- Most protocols between systems, between humans and machines, between components of a system.
- Device drivers such as for tapes and discs that have complicated retry and recovery procedures if the action depends on thestate.

Whenever a feature is directly and explicitly implemented as one or more state transitiontables.

GRAPH MATRICES AND APPLICATIONS

Problem with Pictorial Graphs

- Graphs were introduced as an abstraction ofsoftwarestructure.
- Whenever a graph is used as a model, sooner or later we trace paths through it- to find a set of covering paths, a set of values that will sensitize paths, the logic function that controls the flow, the processing time of the routine, the equations that define the domain, or whether a state is reachable ornot.
- Path is not easy, and it's subject to error. You can miss a link here and there or cover some links twice.
- One solution to this problem is to represent the graph as a matrix and to use matrix operations equivalent to path tracing. These methods are more methodical and mechanical and don't depend on your ability to see a path they aremorereliable.

Tool Building

- If you build test tools or want to know how they work, sooner or later you will be implementing or investigating analysis routines based on thesemethods.
- It is hard to build algorithms over visual graphs so the properties or graph matrices are fundamental to toolbuilding.

The Basic Algorithms

- The basic tool kit consistsof:
- Matrix multiplication, which is used to get the path expression from every node toevery othernode.
- Apartitioningalgorithmforconvertinggraphswithloopsintoloopfreegraphsor equivalence classes.
- A collapsing process which gets the path expression from any node to any othernode.

The Matrix of a Graph

- A graph matrix is a square array with one row and one column for every node inthegraph.
- Each row-column combination corresponds to a relation between the node corresponding to the row and the node corresponding tothecolumn.
- The relation for example, could be as simple as the link name, if there is a link betweenthe nodes.

- Some of the things to beobserved:
- The size of the matrix equals the number ofnodes.
- There is a place to put every possible direct connection or link between any and any othernode.
- The entry at a row and column intersection is the link weight of the link that connects thetwo nodes in thatdirection.
- Aconnectionfromnode itojdoesnotimplyaconnectionfromnode jtonodei.
- If there are several links between two nodes, then the entry is a sum; the "+" sign denotes parallel links asusual.

A simple weight

- A simplest weight we can use is to note that there is or isn't a connection. Let "1" mean that there is a connection and "0" mean that thereisn't.
- The arithmetic rulesare:
- 1+1=1 1*1=1
- 1+0=1 1*0=0
- 0+0=0 0*0=0
- A matrix defined like this is called connectionmatrix.

Connection matrix

- The connection matrix is obtained by replacing each entry with 1 if there is a link and 0 ifthere isn't.
- As usual we don't write down 0 entries to reduce theclutter.

		a	c
	b		
	d		

		1	1
	1		
	1		

Connection Matrix-continued

- Each row of a matrix denotes the out links of the node corresponding to thatrow.
- Each column denotes the in links corresponding to thatnode.
- A branch is a node with more than one nonzero entry in itsrow.
- A junction is node with more than one nonzero entry in itscolumn.
- A self loop is an entry along thediagonal.

Cyclomatic Complexity

- The cyclomatic complexity obtained by subtracting 1 from the total number of entries in each row and ignoring rows with no entries, we obtain the equivalent number of decisions for each row. Addingthesevaluesandthenadding1tothesumyieldsthegraph'scyclomaticcomplexity.

1+1=2 (cyclomatic complexity)

Relations

- A relation is a property that exists between two objects ofinterest.
- Forexample,
- “Node a is connected to node b” or aRb where “R” means “is connectedto”.
- “a>=b” or aRb where “R” means greater than orequal”.
- A graph consists of set of abstract objects called nodes and a relation R between thenodes.
- If aRb, which is to say that a has the relation R to b, it is denoted by a link from a tob.
- For some relations we can associate properties called as linkweights.

Transitive Relations

- A relation is transitive if aRb and bRc impliesaRc.
- Most relations used in testing aretransitive.
- Examples of transitive relations include: is connected to, is greater than or equal to, is less than or equal to, is a relative of, is faster than, is slower than, takes more time than, is a subset of, includes, shadows, is the bossof.
- Examples of intransitive relations include: is acquainted with, is a friend of, is a neighbor of, is lied to, has a du chainbetween.

Reflexive Relations

- A relation R is reflexive if, for every a,aRa.
- A reflexive relation is equivalent to a self loop at everynode.
- Examples of reflexive relations include: equals, is acquainted with, is a relativeof.
- Examples of irreflexive relations include: not equals, is a friend of, is on top of, isunder.

Symmetric Relations

- A relation R is symmetric if for every a and b, aRb impliesbRa.
- A symmetric relation mean that if there is a link from a to b then there is also a link from btoa.
- A graph whose relations are not symmetric are calleddirectedgraph.

- A graph over a symmetric relation is called an undirectedgraph.
- The matrix of an undirected graph is symmetric (aij=aji) for alli,j)

Antisymmetric Relations

- A relation R is antisymmetric if for every a and b, if aRb and bRa, then a=b, or they are the same elements.
- Examples of antisymmetric relations: is greater than or equal to, is a subset of,time.
- Examplesofnonantisymmetricrelations:isconnectedto, canbereachedfrom, isgreaterthan,is a relative of, is a friendof

quivalence Relations

- Anequivalencerelationisarelationthat satisfiesthereflexive, transitive, andsymmetric properties.
- Equality is the most familiar example of an equivalencerelation.
- If a set of objects satisfy an equivalence relation, we say that they form an equivalence class over thatrelation.
- The importance of equivalence classes and relations is that any member of the equivalence class is, with respect to the relation, equivalent to any other member of thatclass.
- Theideabehindpartitiontestingstrategies suchasdomaintestingandpathtesting,isthatwecan partition the input space into equivalenceclasses.
- Testing any member of the equivalence class is as effective as testing themall.

Partial Ordering Relations

- A partial ordering relation satisfies the reflexive, transitive, andantisymmetricproperties.
- Partial ordered graphs have several important properties: they are loop free, there is at least one maximum element, and there is at least one minimumelement.

The Powers of a Matrix

- Each entry in the graph's matrix expresses a relation between the pair of nodes that corresponds to thatentry.
- Squaring the matrix yields a new matrix that expresses the relation between each pair of nodes via one intermediate node under the assumption that the relation istransitive.
- The square of the matrix represents all path segments two linkslong.
- The third power represents all path segments three linkslong.

Matrix Powers and Products

- Given a matrix whose entries are aij, the square of that matrix is obtained by replacingevery entrywith
    - n
    - $a_{ij}=\Sigma a_{ik}a_{kj}$
    - k=1
- more generally, given two matrices A and B with entries aik andbkj, respectively, their product is a new matrix C, whose entries are cij,where:
    - n
    - $C_{ij}=\Sigma a_{ik}b_{kj}$
    - k=1

Partitioning Algorithm

- Consider any graph over a transitive relation. The graph may haveloops.
- Wewouldliketopartitionthegraphbygroupingnodesinsuchawaythateveryloopiscontained within one group oranother.
- Such a graph is partiallyordered.
- There are many used for an algorithm thatdoesthat:
- We might want to embed the loops within a subroutine so as to have a resulting graph which is loop free at the toplevel.
- Many graphs with loops are easy to analyze if you know where to breaktheloops.
- While you and I can recognize loops, it's much harder to program a tool to do it unless you have a solid algorithm on which to basethetool.

Node Reduction Algorithm (General)

- The matrix powers usually tell us more than we want to know about mostgraphs.
- In the context of testing, we usually interested in establishing a relation between twonodes- typically the entry and exitnodes.
- In a debugging context it is unlikely that we would want to know the path expressionbetween every node and every other node.
- Theadvantageofmatrix reduction methodis thatit ismoremethodicalthanthegraphicalmethod called as node by noderemovalalgorithm.

1. Select anodeforremoval;replacethenodebyequivalentlinks thatbypassthatnodeandadd those links to the links theyparallel.
2. Combine the parallel terms and simplify as youcan.
3. Observe loop terms and adjust the out links of every node that had a self loop to account for the effect of theloop.
4. Theresultisamatrixwhosesizehasbeenreducedby1.Continueuntilonlythetwonodesof interestexist.

PART A(2 MARKS)

1. Define the term Projectmonitoring.

   Project Monitoring refers to the activities and tasks managers engage in to periodically check the status of each project. Reports are prepared that compare the actual work done to the work that was planned**.**

2. Define the term Projectcontrolling.

   Project Controlling consists of developing and applying a set of corrective actions to get a project on track when monitoring shows a deviation from what was planned.

3. Define Milestones

   Milestones are tangible events that are expected to occur at a certain time in the project's lifetime. Managers use them to determine project status.

4. List some examples of testing Milestones

   Completion of the Master test plan

   Completion of branch coverage for allunits

   Execution of all planned system test

   Completion of the test summaryreport.

5. List various Measurements for monitoring testingstatus.
   Coverage Measures

   Test Case Development

   Test Execution

   Test Harness Development

6. List the types of testing measurementsCoverage

   Test CaseDevelopment

   Test Execution

   Test Harness

7. What are the various Severity levelhierarchy

   Catastrophic

   Critical

   Marginal

   Minor or Annoying

8. What are the four major activities associated with Configuration management.

   Identification of the Configurationitems

   Change Control 9 Configuration status reporting

   Configuration audits

9. Define Change Control Board(CCB).

   There are 2 aspects of change control – one is tool based, the other term based. The team involved is called CCB.

10. Define the termReview.

    A review is a group meeting whose purpose is to evaluate a software artifact or a set of software

artifact.

11. Explain the benefits of

reviewprogramHigher –

qualitysoftware

Increased productivity

Closer adherence to project schedule

Increased awareness of quality issues

12. List the types ofreviews.

There are two major types of technical reviews

Inspections

Walkthrough

13. What are the various steps in the inspection process

EntryCriteria

Initiation

Preparation

Inspection Meeting

Reporting results

Rework & follow up

14. DefineWalkthrough

Walkthrough are a type of technical review where the producer of the reviewed material serves as the review leader and actually guides the progression of the review. Walkthrough have traditionally been applied to design and code.

15. What are the advantages of review approach.

There are two pass approach for detect detection.

Pass 1 has individuals first reading reviewed item

Pass 2 has the item read by the group as awhole.

16. What are the various components of reviewplans.

Review Goals

Preconditions and items to be reviewed

Roles, Participants, Team Size and time requirements

Review Procedures

Review Training

Review Checklist

17. What are the various roles in review program

Review Leader

Review Recorder

Reader

Reviewer

18. List the various review team membership constituency Review TeamMembers

SQAStaff

Testers

Developers

Users / Clients

Specialists

19. What are the various different types of softwareartifacts.

RequirementReviews

Design Reviews

Code Reviews

Test Plan reviews

20. Define Defect Removal Leverage(DRL).

This is a ratio of the defect detection rates from two review or test phases and can be expressed as

$$DRL = \frac{\text{Defects / hour (review or test phase X)}}{\text{Defects / hour (review or test phase Y)}}$$

PART B(10 MARKS)

1. The behaviour of a finite state machine is invariant under all encodings.Justify
2. Discuss node reduction algorithm for graphmatrices.
3. What are the advantages and disadvantages of arrayrepresentations.
4. Define graph matrices andtheirapplications?
5. Discuss a node reduction algorithm in terms ofmatrixoperations?
6. Demonstrate an algorithm for nodereduction(general)?
7. Illustrate the applications of nodereductionalgorithm?
8. Demonstrate design guidelines for building finite state machines into yourcode?
9. Illustrate designer’s comments aboutstategraphs?
10. Draw a hard disk recovery a state graph with astatetable**?**

SOFTWARE TESTING METHODOLOGIES

BY

Dr. Vikas Panthi

Software Testing Methodologies

Text Books: 1.Software Testing Techniques: Boris Beizer
2. Craft of Software Testing: Brain Marrick

UNIT-I

Introduction: Purpose of testing. Dichotomies. Model for testing.

Consequences of bugs. Taxonomy of bugs.

Flow graphs and Path testing: Basics concepts of path testing, predicates, path predicates and achievable paths, path sensitizing, path instrumentation, application of path testing.

Introduction

What is Testing?

Related terms : SQA, QC, Verification, Validation

Verification of functionality for conformation against given specifications
By execution of the software application

A Test

Passes: Functionality OK.
Fails: Application functionality NOK.
Bug/Defect/Fault: Deviation from expected functionality.
It’s not always obvious.

Purpose of Testing

1. To Catch Bugs

   - Bugs are due to imperfect Communication among programmers
     - Specs, design, low level functionality
   - Statistics say: about 3 bugs / 100 statements

2. Productivity Related Reasons

   - Insufficient effort in QA => High Rejection Ratio => Higher Rework => Higher Net Costs

   - Statistics:
     - QA costs: 2% for consumer products
       80% for critical software

   - Quality ⇔ Productivity

Purpose of Testing

Purpose of testing contd…

3. Goals for testing

Primary goal of Testing: Bug Prevention

- Bug prevented ⇒ rework effort is saved [bug reporting, debugging, correction, retesting]
- If it is not possible, Testing must reach its secondary goal of bud discovery.
- Good test design & tests ⇒ clear diagnosis ⇒ easy bug correction

Test Design Thinking

- From the specs, write test specs. First and then code.
- Eliminates bugs at every stage of SDLC.
- If this fails, testing is to detect the remaining bugs.

4. 5 Phases in tester's thinking

Phase 0: says no difference between debugging & testing

- Today, it's a barrier to good testing & quality software.

Purpose of Testing

Purpose of testing contd…

Phase 1: says Testing is to show that the software works

- A failed test shows software does not work, even if many tests pass.
- Objective not achievable.

Phase 2: says Software does not work

- One failed test proves that.
- Tests are to be redesigned to test corrected software.
- But we do not know when to stop testing.

Phase 3: says Test for Risk Reduction

- We apply principles of statistical quality control.
- Our perception of the software quality changes – when a test passes/fails.
- Consequently, perception of product Risk reduces.
- Release the product when the Risk is under a predetermined limit.

Purpose of Testing

5 Phases in tester's thinking continued…

Phase 4: A state of mind regarding "What testing can do & cannot do. What makes software testable".

- Applying this knowledge reduces amount of testing.
- Testable software reduces effort
- Testable software has less bugs than the code hard to test

Cumulative goal of all these phases:

- Cumulative and complementary. One leads to the other.
- Phase2 tests alone will not show software works
- Use of statistical methods to test design to achieve good testing at acceptable risks.
- Most testable software must be debugged, must work, must be hard to break.

Purpose of Testing

purpose of testing contd..

5. Testing & Inspection

- Inspection is also called static testing.
- Methods and Purposes of testing and inspection are different, but the objective is to catch & prevent different kinds of bugs.
- To prevent and catch most of the bugs, we must

  - Review
  - Inspect &
  - Read the code

  - Do walkthroughs on the code

& then do Testing

Purpose of Testing

Further…

Some important points:

Test Design After testing & corrections, Redesign tests & test the redesigned tests

Bug Prevention

Mix of various approaches, depending on factors

culture, development environment, application, project size, history, language

Inspection Methods

Design Style

Static Analysis

Languages – having strong syntax, path verification & other controls

Design methodologies & development environment

Its better to know:

Pesticide paradox

Complexity Barrier

Dichotomies

- division into two especially mutually exclusive or contradictory groups or entities
- the *dichotomy* between theory and practice

Let us look at six of them:

1. Testing & Debugging
2. Functional Vs Structural Testing
3. Designer vs Tester
4. Modularity (Design) vs Efficiency
5. Programming in SMALL Vs programming in BIG
6. Buyer vs Builder

Dichotomies

1. Testing Vs Debugging

- Testing is to find bugs.
- Debugging is to find the cause or misconception leading to the bug.

- Their roles are confused to be the same. But, there are differences in goals, methods and psychology applied to these

#	Testing	Debugging
1	Starts with known conditions. Uses predefined procedure. Has predictable outcomes.	Starts with possibly unknown initial conditions. End cannot be predicted.
2	Planned, Designed and Scheduled.	Procedures & Duration are not constrained.
3	A demo of an error or apparent correctness.	A Deductive process.
4	Proves programmer's success or failure.	It is programmer's Vindication.
5	Should be predictable, dull, constrained, rigid & inhuman.	There are intuitive leaps, conjectures, experimentation & freedom.

Dichotomies

Dichotomies contd…

#	Testing	Debugging
6	Much of testing can be without design knowledge.	Impossible without a detailed design knowledge.
7	Can be done by outsider to the development team.	Must be done by an insider (development team).
8	A theory establishes what testing can do or cannot do.	There are only Rudimentary Results (on how much can be done. Time, effort, how etc. depends on human ability).
9	Test execution and design can be automated.	Debugging - Automation is a dream.

Dichotomies

Dichotomies contd..

2. Functional Vs Structural Testing

- **Functional Testing:** Treats a program as a black box. Outputs are verified for conformance to specifications from user's point of view.

- **Structural Testing:** Looks at the implementation details: programming style, control method, source language, database & coding details.

- **Interleaving of functional & Structural testing:**
  - A good program is built in layers from outside.
  - Outside layer is pure system function from user's point of view.
  - Each layer is a structure with its outer layer being its function.
  - Examples:

Dichotomies

- **Interleaving of functional & Structural testing: (contd..)**
  - For a given model of programs, Structural tests may be done first and later the Functional, Or vice-versa. Choice depends on which seems to be the natural choice.
  - Both are useful, have limitations and target different kind of bugs. Functional tests can detect all bugs in principle, but would take infinite amount of time. Structural tests are inherently finite, but cannot detect all bugs.
  - The Art of Testing is how much allocation % for structural vs how much % for functional.

Dichotomies

Dichotomies contd..

3. Designer vs Tester

- ✓ Completely separated in black box testing. Unit testing may be done by either.
- ✓ Artistry of testing is to balance knowledge of design and its biases against ignorance & inefficiencies.
- ✓ Tests are more efficient if the designer, programmer & tester are independent in all of unit, unit integration, component, component integration, system, formal system feature testing.
- ✓ The extent to which test designer & programmer are separated or linked depends on testing level and the context.

#	Programmer / Designer	Tester
1	Tests designed by designers are more oriented towards structural testing and are limited to its limitations.	With knowledge about internal test design, the tester can eliminate useless tests, optimize & do an efficient test design.
2	Likely to be biased.	Tests designed by independent testers are bias-free.
3	Tries to do the job in simplest & cleanest way, trying to reduce the complexity.	Tester needs to suspicious, uncompromising, hostile and obsessed with destroying program.

Dichotomies

Dichotomies contd..

4. Modularity (Design) vs Efficiency

1. system and test design can both be modular.
2. A module implies a size, an internal structure and an interface, Or, in other words.
3. A module (well defined discrete component of a system) consists of internal complexity & interface complexity and has a size.

Dichotomies

#	Modularity	Efficiency
1	Smaller the component easier to understand.	Implies more number of components & hence more # of interfaces increase complexity & reduce efficiency (=> more bugs likely)
2	Small components/modules are repeatable independently with less rework (to check if a bug is fixed).	Higher efficiency at module level, when a bug occurs with small components.
3	Microscopic test cases need individual setups with data, systems & the software. Hence can have bugs.	More # of test cases implies higher possibility of bugs in test cases. Implies more rework and hence less efficiency with microscopic test cases
4	Easier to design large modules & smaller interfaces at a higher level.	Less complex & efficient. (Design may not be enough to understand and implement. It may have to be broken down to implementation level.)

So:

- Optimize the size & balance internal & interface complexity to increase efficiency

- Optimize the test design by setting the scopes of tests & group of tests (modules) to minimize cost of test design, debugging, execution & organizing – without compromising effectiveness.

Dichotomies

Dichotomies contd..

5. Programming in SMALL Vs programming in BIG

- ✓ Impact on the development environment due to the volume of customer requirements.

#	Small	Big
1	More efficiently done by informal, intuitive means and lack of formality – if it's done by 1 or 2 persons for small & intelligent user population.	A large # of programmers & large # of components.
2	Done for e.g., for oneself, for one's office or for the institute.	Program size implies non-linear effects (on complexity, bugs, effort, rework quality).
3	Complete test coverage is easily done.	Acceptance level could be: Test coverage of 100% for unit tests and for overall tests ≥ 80%.

Dichotomies

6. Buyer Vs Builder (customer vs developer organization)

- Buyer & Builder being the same (organization) clouds accountability.
- Separate them to make the accountability clear, even if they are in the same organization.
- The accountability increases motivation for quality.

- The roles of all parties involved are:

- Builder:
  - Designs for & is accountable to the Buyer.
- Buyer:
  - Pays for the system.
  - Hopes to get profits from the services to the User.
- User:
  - Ultimate beneficiary of the system.
  - Interests are guarded by the Tester.
- Tester:
  - *Dedicated to the destruction of the s/w (builder)*
  - Tests s/w in the interests of User/Operator.
- Operator:
  - Lives with: Mistakes of the Builder, Oversights of Tester, Murky specs of Buyer, Complaints of User

- A model for testing - with a project environment - with tests at various levels.
- (1) understand what a project is. (2) look at the roles of the Testing models.

1. PROJECT:

- An Archetypical System (product) allows tests without complications (even for a large project).
- Testing a one shot routine & very regularly used routine is different.

- A model for project in a real world consists of the following 8 components:

1) Application: An online real-time system (with remote terminals) providing timely responses to user requests (for services).

2) Staff: Manageable size of programming staff with specialists in systems design.

3) Schedule: project may take about 24 months from start to acceptance. 6 month maintenance period.

4) Specifications: is good. documented. Undocumented ones are understood well in the team.

A Model for Testing

4) Acceptance test: Application is accepted after a formal acceptance test. At first it's the customer's & then the software design team's responsibility.

5) Personnel: The technical staff comprises of : A combination of experienced professionals & junior programmers (1 – 3 yrs) with varying degrees of knowledge of the application.

6) Standards:
   - Programming, test and interface standard (documented and followed).
   - A centralized standards data base is developed & administrated

A Model for Testing

1. PROJECT: contd …

6) Objectives: (of a project)

- A system is expected to operate profitably for > 10 yrs (after installation).
- Similar systems with up to 75% code in common may be implemented in future.

7) Source: (for a new project) is a combination of

- New Code - up to 1/3rd
- From a previous reliable system - up to 1/3rd
- Re-hosted from another language & O.S. - up to 1/3rd

8) History: Typically:

- Developers quit before his/her components are tested.
- Excellent but poorly documented work.
- Unexpected changes (major & minor) from customer may come in
- Important milestones may slip, but the delivery date is met.
- Problems in integration, with some hardware, redoing of some component etc…..

A model project is

☺ A Well Run & Successful Project.
😐 Combination of Glory and Catastrophe.

A Model for Testing

A Model for Testing contd..

2. Roles of Models for Testing

1) Overview:

- Testing process starts with a program embedded in an environment.
- Human nature of susceptibility to error leads to 3 models.
- Create tests out of these models & execute
- Results is expected ⇒ It's okay
  unexpected ⇒ Revise tests and program. Revise bug model and program.

2) Environment: includes

- All hardware & software (firmware, OS, linkage editor, loader, compiler, utilities, libraries) required to make the program run.
- Usually bugs do not result from the environment. (with established h/w & s/w)
- But arise from our understanding of the environment.

3) Program:

- Complicated to understand in detail.
- Deal with a simplified overall view.
- Focus on control structure ignoring processing & focus on processing ignoring control structure.
- If bug's not solved, modify the program model to include more facts, & if that fails, modify the program.

A Model for Testing contd..

2. Roles of Models for Testing contd …

4) Bugs: (bug model)

- Categorize the bugs as initialization, call sequence, wrong variable etc..
- An incorrect spec. may lead us to mistake for a program bug.
- There are 9 Hypotheses regarding Bugs.

a. Benign Bug Hypothesis:

- The belief that the bugs are tame & logical.
- Weak bugs are logical & are exposed by logical means.
- Subtle bugs have no definable pattern.

b. Bug locality hypothesis:

- Belief that bugs are localized.
- Subtle bugs affect that component & external to it.

c. Control Dominance hypothesis:

- Belief that most errors are in control structures, but data flow & data structure errors are common too.
- Subtle bugs are not detectable only thru control structure.

(subtle bugs => from violation of data structure boundaries & data-code separation)

A Model for Testing contd..

2. Roles of Models for Testing contd …

4) Bugs: (bug model) contd ..

d. Code/data Separation hypothesis:

- Belief that the bugs respect code & data separation in HOL programming.
- In real systems the distinction is blurred and hence such bugs exist.

e. Lingua Salvator Est hypothesis:

- Belief that the language syntax & semantics eliminate most bugs.
- But, such features may not eliminate Subtle Bugs.

f. Corrections Abide hypothesis:

- Belief that a corrected bug remains corrected.
- Subtle bugs may not. For e.g.
  A correction in a data structure ‘DS’ due to a bug in the interface between modules A & B, could impact module C using ‘DS’.

A Model for Testing contd..

2. Roles of Models for Testing contd …

4) Bugs: (bug model) contd ..

g. Silver Bullets hypothesis:

- Belief that - language, design method, representation, environment etc. grant immunity from bugs.
- Not for subtle bugs.
- *Remember the pesticide paradox.*

h. Sadism Suffices hypothesis:

- Belief that a sadistic streak, low cunning & intuition (by independent testers) are sufficient to extirpate most bugs.
- Subtle & tough bugs are may not be … - these need methodology & techniques.

i. Angelic Testers hypothesis:

- Belief that testers are better at test design than programmers at code design.

A Model for Testing contd..

2. Roles of Models for Testing contd..

5) Tests:

- Formal procedures.
- Input preparation, outcome prediction and observation, documentation of test, execution & observation of outcome are subject to errors.
- An unexpected test result may lead us to revise the test and test model.

6) Testing & Levels:

3 kinds of tests (with different objectives)

1) Unit & Component Testing

a. A unit is the smallest piece of software that can be compiled/assembled, linked, loaded & put under the control of test harness / driver.

b. Unit testing - verifying the unit against the functional specs & also the implementation against the design structure.

c. Problems revealed are unit bugs.

d. Component is an integrated aggregate of one or more units (even entire system)

e. Component testing - verifying the component against functional specs and the implemented structure against the design.

f. Problems revealed are component bugs.

A Model for Testing contd..

2. Roles of Models for Testing contd …

2) Integration Testing:

- Integration is a process of aggregation of components into larger components.
- Verification of consistency of interactions in the combination of components.
- Examples of integration testing are improper call or return sequences, inconsistent data validation criteria & inconsistent handling of data objects.

- Integration testing & Testing Integrated Objects are different

- Sequence of Testing:
  - Unit/Component tests for A, B. Integration tests for A & B. Component testing for (A,B) component

A Model for Testing contd..

2. Roles of Models for Testing contd …

3) System Testing

a. System is a big component.
b. Concerns issues & behaviors that can be tested at the level of entire or major part of the integrated system.
c. Includes testing for performance, security, accountability, configuration sensitivity, start up & recovery

After understanding a Project, Testing Model, now let's see finally,

Role of the Model of testing :

- Used for the testing process until system behavior is correct or until the model is insufficient (for testing).
- Unexpected results may force a revision of the model.
- Art of testing consists of creating, selecting, exploring and revising models.
- The model should be able to express the program.

Consequences of Bugs

Consequences: (how bugs may affect users)

These range from mild to catastrophic on a 10 point scale.

- Mild
  - Aesthetic bug such as misspelled output or mal-aligned print-out.
- Moderate
  - Outputs are misleading or redundant impacting performance.
- Annoying
  - Systems behavior is dehumanizing for e.g. names are truncated/modified arbitrarily, bills for $0.0 are sent.
  - Till the bugs are fixed operators must use unnatural command sequences to get proper response.
- Disturbing
  - Legitimate transactions refused.
  - For e.g. ATM machine may malfunction with ATM card / credit card.
- Serious
  - Losing track of transactions & transaction events. Hence accountability is lost.

Consequences of Bugs

Consequences contd …

- Very serious
  System does another transaction instead of requested e.g. Credit another account, convert withdrawals to deposits.

- Extreme
  - Frequent & Arbitrary - not sporadic & unusual.

- Intolerable
  - Long term unrecoverable corruption of the Data base.
    (not easily discovered and may lead to system down.)

- Catastrophic
  - System fails and shuts down.

- Infectious
  - Corrupts other systems, even when it may not fail.

Consequences of Bugs

Consequences contd …

Assignment of severity

- Assign flexible & relative rather than absolute values to the bug (types).
- Number of bugs and their severity are factors in determining the quality quantitatively.
- Organizations design & use quantitative, quality metrics based on the above.

- Parts are weighted depending on environment, application, culture, correction cost, current SDLC phase & other factors.

- Nightmares

  - Define the nightmares – that could arise from bugs – for the context of the organization/application.

  - Quantified nightmares help calculate importance of bugs.
  - That helps in making a decision on when to stop testing & release the product.

Consequences of Bugs

Consequences contd …

When to stop Testing

1. List all nightmares in terms of the symptoms & reactions of the user to their consequences.
2. Convert the consequences of into a cost. There could be rework cost. *(but if the scope extends to the public, there could be the cost of lawsuits, lost business, nuclear reactor meltdowns.)*
3. Order these from the costliest to the cheapest. Discard those with which you can live with.
4. Based on experience, measured data, intuition, and published statistics postulate the kind of bugs causing each symptom. This is called 'bug design process'. A bug type can cause multiple symptoms.
5. Order the causative bugs by decreasing probability *(judged by intuition, experience, statistics etc.)*. Calculate the importance of a bug type as:

$$\text{Importance of bug type } j = \sum_{\text{all } k} C_{jk} \; P_{jk}$$

where,

C_{jk} = cost due to bug type j causing nightmare k

P_{jk} = probability of bug type j causing nightmare k

$$\left(\text{Cost due to all bug types} = \sum_{\text{all } k} \sum_{\text{all } j} C_{jk} \; P_{jk} \right)$$

Consequences of Bugs

Consequences contd … When to stop Testing contd ..

6. Rank the bug types in order of decreasing importance.

7. Design tests & QA inspection process with most effective against the most important bugs.

8. If a test is passed or when correction is done for a failed test, some nightmares disappear. As testing progresses, revise the probabilities & nightmares list as well as the test strategy.

9. Stop testing when probability (importance & cost) proves to be inconsequential.

This procedure could be implemented formally in SDLC.

Important points to Note:

- Designing a reasonable, finite # of tests with high probability of removing the nightmares.
- Test suites wear out.
  - As programmers improve programming style, QA improves.
  - Hence, know and update test suites as required.

Taxonomy of Bugs ..

we had seen the:

1. Importance of Bugs - statistical quantification of impact
2. Consequences of Bugs - causes, nightmares, to stop testing

We will now see the:

3. Taxonomy of Bugs - along with some remedies

In order to be able to create an organization's own Bug Importance Model for the sake of controlling associated costs…

Taxonomy of Bugs .. and remedies

Reference of IEEE Taxonomy: IEEE 87B

- Why Taxonomy ?
  To study the consequences, nightmares, probability, importance, impact and the methods of prevention and correction.

- Adopt known taxonomy to use it as a statistical framework on which your testing strategy is based.

- 6 main categories with sub-categories..

1)Requirements, Features, Functionality Bugs	**24.3% bugs**
2)Structural Bugs	**25.2%**
3)Data Bugs	**22.3%**
4)Coding Bugs	**9.0%**
5)Interface, Integration and System Bugs	**10.7%**
6)Testing & Test Design Bugs	**2.8 %**

Taxonomy of Bugs .. and remedies

Reference of IEEE Taxonomy: IEEE 87B

1) Requirements, Features, Functionality Bugs

3 types of bugs : **Requirement & Specs, Feature, & feature interaction bugs**

I. Requirements & Specs.

- Incompleteness, ambiguous or self-contradictory
- Analyst's assumptions not known to the designer

- Some thing may miss when specs change
- These are expensive: introduced early in SDLC and removed at the last

II. Feature Bugs

- Specification problems create feature bugs
- Wrong feature bug has design implications

- Missing feature is easy to detect & correct
- Gratuitous enhancements can accumulate bugs, if they increase complexity

- Removing features may foster bugs

Taxonomy of Bugs .. and remedies

1) Requirements, Features, Functionality Bugs contd..

III. Feature Interaction Bugs

- Arise due to unpredictable interactions between feature groups or individual features. The earlier removed the better as these are costly if detected at the end.
- Examples: call forwarding & call waiting. Federal, state & local tax laws.
- No magic remedy. Explicitly state & test important combinations

Remedies

- Use high level formal specification languages to eliminate human-to-human communication
- It's only a short term support & not a long term solution

- Short-term Support:
  - Specification languages formalize requirements & so automatic test generation is possible. It's cost-effective.

- Long-term support:
  - Even with a great specification language, problem is not eliminated, but is shifted to a higher level. Simple ambiguities & contradictions may only be removed, leaving tougher bugs.

Testing Techniques

- Functional test techniques - **transaction flow testing, syntax testing, domain testing, logic testing, and state testing** - can eliminate requirements & specifications bugs.

Taxonomy of Bugs .. and remedies

2. Structural Bugs

we look at the 5 types, their causes and remedies.

I. Control & Sequence bugs
II. Logic Bugs
III. Processing bugs
IV. Initialization bugs
V. Data flow bugs & anomalies

1. Control & Sequence Bugs:

- Paths left out, unreachable code, spaghetti code, and pachinko code.
- Improper nesting of loops, Incorrect loop-termination or look-back, ill-conceived switches.
- Missing process steps, duplicated or unnecessary processing, rampaging GOTOs.
- Novice programmers.
- Old code (assembly language & Cobol)

Prevention and Control:

- Theoretical treatment and,
- Unit, structural, path, & functional testing.

Taxonomy of Bugs .. and remedies

2. Structural Bugs contd..

II. Logic Bugs

- Misunderstanding of the semantics of the control structures & logic operators
- Improper layout of cases, including impossible & ignoring necessary cases,

- Using a look-alike operator, improper simplification, confusing Ex-OR with inclusive OR.
- Deeply nested conditional statements & using many logical operations in 1 stmt.

Prevention and Control:

Logic testing, careful checks, functional testing

III. Processing Bugs

- Arithmetic, algebraic, mathematical function evaluation, algorithm selection & general. processing, data type conversion, ignoring overflow, improper use of relational operators.

Prevention

- Caught in Unit Testing & have only localized effect
- Domain testing methods

Taxonomy of Bugs .. and remedies

Structural bugs contd..

IV. Initialization Bugs

- Forgetting to initialize work space, registers, or data areas.
- Wrong initial value of a loop control parameter.

- Accepting a parameter without a validation check.
- Initialize to wrong data type or format.
- Very common.

Remedies (prevention & correction)

- Programming tools, Explicit declaration & type checking in source language, preprocessors.
- Data flow test methods help design of tests and debugging.

V. Dataflow Bugs & Anomalies

- Run into an un-initialized variable.
- Not storing modified data.

- Re-initialization without an intermediate use.
- Detected mainly by execution (testing).

Remedies (prevention & correction)

- Data flow testing methods & matrix based testing methods.

Taxonomy of Bugs .. and remedies

3. Data Bugs

Depend on the types of data or the representation of data. There are 4 sub categories.

I. Generic Data Bugs

II. Dynamic Data Vs Static Data

III. Information, Parameter, and Control Bugs

IV. Contents, Structure & Attributes related Bugs

Taxonomy of Bugs .. and remedies

Data Bugs contd…

I. Generic Data Bugs

- Due to data object specs., formats, # of objects & their initial values.
- Common as much as in code, especially as the code migrates to data.
- Data bug introduces an operative statement bug & is harder to find.
- Generalized components with reusability – when customized from a large parametric data to specific installation.

Remedies (prevention & correction):

- Using control tables in lieu of code facilitates software to handle many transaction types with fewer data bugs. Control tables have a hidden programming language in the database.
- Caution - there's no compiler for the hidden control language in data tables

Taxonomy of Bugs .. and remedies

II. Dynamic Data Vs Static Data

Dynamic Data Bugs	Static Data Bugs
Transitory. Difficult to catch.	Fixed in form & content.
Due to an error in a shared storage object initialization.	Appear in source code or data base, directly or indirectly
Due to unclean / leftover garbage in a shared resource.	Software to produce object code creates a static data table – bugs possible
Examples	**Examples**
Generic & shared variable	Telecom system software: generic parameters, a generic large program & site adapter program to set parameter values, build data declarations etc.
Shared data structure	Postprocessor : to install software packages. Data is initialized at run time – with configuration handled by tables.
Prevention Data validation, unit testing	**Prevention** Compile time processing Source language features

Taxonomy of Bugs .. and remedies

Data Bugs contd..

III. Information, Parameter, and Control Bugs

Static or dynamic data can serve in any of the three forms. It is a matter of perspective.

What is information can be a data parameter or control data else where in a program.

Examples: name, hash code, function using these. A variable in different contexts.

- **Information:** dynamic, local to a single transaction or task.
- **Parameter**: parameters passed to a call.
- **Control**: data used in a control structure for a decision.

Bugs

- Usually simple bugs and easy to catch.
- When a subroutine (with good data validation code) is modified, forgetting to update the data validation code, results in these bugs.

Preventive Measures (prevention & correction)

- Proper Data validation code.

Taxonomy of Bugs .. and remedies

Data Bugs contd..

IV. Contents, Structure & Attributes related Bugs

- **Contents**: are pure bit pattern & bugs are due to misinterpretation or corruption of it.
- **Structure**: Size, shape & alignment of data object in memory. A structure may have substructures.
- **Attributes**: Semantics associated with the contents (e.g. integer, string, subroutine).

Bugs

- Severity & subtlety increases from contents to attributes as they get less formal.
- Structural bugs may be due to wrong declaration or when same contents are interpreted by multiple structures differently (different mapping).

- Attribute bugs are due to misinterpretation of data type, probably at an interface

Preventive Measures (prevention & correction)

- Good source lang. documentation & coding style (incl. data dictionary).
- Data structures be globally administered. Local data migrates to global.

- Strongly typed languages prevent mixed manipulation of data.
- In an assembly lang. program, use field-access macros & not directly accessing any field.

4. Coding Bugs

- Coding errors create other kinds of bugs.
- Syntax errors are removed when compiler checks syntax.
- Coding errors
  typographical, misunderstanding of operators or statements or could be just arbitrary.
- Documentation Bugs
  - Erroneous comments could lead to incorrect maintenance.
  - Testing techniques cannot eliminate documentation bugs.
  - Solution:

    Inspections, QA, automated data dictionaries & specification systems.

Taxonomy of Bugs .. and remedies

5. Interface, Integration and Systems Bugs

There are 9 types of bugs of this type.

1) External Interfaces
2) Internal Interfaces
3) Hardware Architecture Bugs
4) Operating System Bugs
5) Software architecture bugs
6) Control & Sequence bugs
7) Resource management bugs
8) Integration bugs
9) System bugs

Taxonomy of Bugs .. and remedies

5. Interface, Integration and Systems Bugs contd..

1) External Interfaces

- Means to communicate with the world: drivers, sensors, input terminals, communication lines.
- Primary design criterion should be - robustness.

- Bugs: invalid timing, sequence assumptions related to external signals, misunderstanding external formats and no robust coding.
- Domain testing, syntax testing & state testing are suited to testing external interfaces.

2) Internal Interfaces

- Must adapt to the external interface.
- Have bugs similar to external interface

- Bugs from improper
  - Protocol design, input-output formats, protection against corrupted data, subroutine call sequence, call-parameters.

- Remedies (prevention & correction):

  - Test methods of domain testing & syntax testing.
  - Good design & standards: good trade off between # of internal interfaces & complexity of the interface.
  - Good integration testing is to test all internal interfaces with external world.

Taxonomy of Bugs .. and remedies

Interface, Integration and Systems Bugs contd …

3) Hardware Architecture Bugs:

- A s/w programmer may not see the h/w layer / architecture.
- S/w bugs originating from hardware architecture are due to misunderstanding of how h/w works.
- **Bugs are due to errors in:**
  - Paging mechanism, address generation
  - I/O device instructions, device status code, device protocol
  - Expecting a device to respond too quickly, or to wait for too long for response, assuming a device is initialized, interrupt handling, I/O device address
  - H/W simultaneity assumption, H/W race condition ignored, device data format error etc..
- Remedies (prevention & correction):
  - Good software programming & Testing.
  - Centralization of H/W interface software.
  - Nowadays hardware has special test modes & test instructions to test the H/W function.
  - An elaborate H/W simulator may also be used.

Taxonomy of Bugs .. and remedies

Interface, Integration and Systems Bugs contd …

4) Operating System Bugs:

- **Due to:**
  - Misunderstanding of H/W architecture & interface by the O. S.
  - Not handling of all H/W issues by the O. S.
  - Bugs in O. S. itself and some corrections may leave quirks.
  - Bugs & limitations in O. S. may be buried some where in the documentation.
- **Remedies (prevention & correction):**
  - Same as those for H/W bugs.
  - Use O. S. system interface specialists
  - Use explicit interface modules or macros for all O.S. calls.
  - The above may localize bugs and make testing simpler.

Taxonomy of Bugs .. and remedies

Interface, Integration and Systems Bugs contd …

5) Software Architecture Bugs: (called Interactive)

The subroutines pass thru unit and integration tests without detection of these bugs. Depend on the Load, when the system is stressed. These are the most difficult to find and correct.

- **Due to:**
  - Assumption that there are no interrupts, Or, Failure to block or unblock an interrupt.
  - Assumption that code is re-entrant or not re-entrant.
  - Bypassing data interlocks, Or, Failure to open an interlock.
  - Assumption that a called routine is memory resident or not.
  - Assumption that the registers and the memory are initialized, Or, that their content did not change.
  - Local setting of global parameters & Global setting of local parameters.
- **Remedies:**
  - Good design for software architecture.
- **Test Techniques**
  - All test techniques are useful in detecting these bugs, Stress tests in particular.

Taxonomy of Bugs .. and remedies

Interface, Integration and Systems Bugs contd …

6) Control & Sequence Bugs:

- **Due to:**
  - Ignored timing
  - Assumption that events occur in a specified sequence.
  - Starting a process before its prerequisites are met.
  - Waiting for an impossible combination of prerequisites.
  - Not recognizing when prerequisites are met.
  - Specifying wrong priority, Program state or processing level.
  - Missing, wrong, redundant, or superfluous process steps.

- **Remedies**:
  - Good design.
  - highly structured sequence control - useful
  - Specialized internal sequence-control mechanisms such as an internal job control language – useful.
  - Storage of Sequence steps & prerequisites in a table and interpretive processing by control processor or dispatcher - easier to test & to correct bugs.

- **Test Techniques**
  - Path testing as applied to Transaction Flow graphs is effective.

Taxonomy of Bugs .. and remedies

Interface, Integration and Systems Bugs contd …

7) Resource Management Problems:

- Resources: Internal: Memory buffers, queue blocks etc. External: discs etc.

- **Due to:**
  - Wrong resource used (when several resources have similar structure or different kinds of resources in the same pool).
  - Resource already in use, or deadlock
  - Resource not returned to the right pool, Failure to return a resource. Resource use forbidden to the caller.

- **Remedies**:
  - Design: keeping resource structure simple with fewest kinds of resources, fewest pools, and no private resource mgmt.
  - Designing a complicated resource structure to handle all kinds of transactions to save memory is not right.
  - Centralize management of all resource pools thru managers, subroutines, macros etc.

- **Test Techniques**
  - Path testing, transaction flow testing, data-flow testing & stress testing.

Taxonomy of Bugs .. and remedies

Interface, Integration and Systems Bugs contd …

8) Integration Bugs:

Are detected late in the SDLC and cause several components and hence are very costly.

- **Due to:**
  - Inconsistencies or incompatibilities between components.
  - Error in a method used to directly or indirectly transfer data between components. Some communication methods are: data structures, call sequences, registers, semaphores, communication links, protocols etc..
- **Remedies**:
  - Employ good integration strategies. ***
- **Test Techniques**
  - Those aimed at interfaces, domain testing, syntax testing, and data flow testing when applied across components.

Taxonomy of Bugs .. and remedies

Interface, Integration and Systems Bugs contd …

9) System Bugs:

- Infrequent, but are costly

- **Due to:**
  - Bugs not ascribed to a particular component, but result from the totality of interactions among many components such as:
    programs, data, hardware, & the O.S.

- **Remedies:**
  - Thorough testing at all levels and the test techniques mentioned below

- **Test Techniques**
  - Transaction-flow testing.
  - All kinds of tests at all levels as well as integration tests - are useful.

6. Testing & Test Design Bugs

Bugs in Testing (scripts or process) are not software bugs.

It's difficult & takes time to identify if a bug is from the software or from the test script/procedure.

1) Bugs could be due to:

- Tests require code that uses complicated scenarios & databases, to be executed.
- Though an independent functional testing provides an un-biased point of view, this lack of bias may lead to an incorrect interpretation of the specs.
- Test Criteria
  - Testing process is correct, but the criterion for judging software's response to tests is incorrect or impossible.
  - If a criterion is quantitative (throughput or processing time), the measurement test can perturb the actual value.

Taxonomy of Bugs .. and remedies

Testing & Test Design Bugs contd…

- Remedies:

1. Test Debugging:

   Testing & Debugging tests, test scripts etc. Simpler when tests have localized affect.

2. Test Quality Assurance:

   To monitor quality in independent testing and test design.

3. Test Execution Automation:

   Test execution bugs are eliminated by test execution automation tools & not using manual testing.

4. Test Design Automation:

   Test design is automated like automation of software development. For a given productivity rate, It reduces bug count.

A word on productivity

At the end of a long study on taxonomy, we could say

Good design inhibits bugs and is easy to test. The two factors are multiplicative and results in high productivity.

Good test works best on good code and good design.

Good test cannot do a magic on badly designed software.

Source: Boris Beizer

Introduction

Path Testing Definition

A family of structural test techniques based on judiciously selecting a set of test paths through the programs.

- ✓ **Goal:** Pick enough paths to assure that every source statement is executed at least once.
- ✓ It is a measure of thoroughness of **code coverage.**
- ✓ It is used most for unit testing on new software.
- ✓ Its effectiveness reduces as the software size increases.
- ✓ We use Path testing techniques indirectly.
- ✓ Path testing concepts are used **in** and **along** with other testing techniques

Code Coverage: During unit testing: # stmts executed at least once / total # stmts

Path Testing contd..

Assumptions:

- Software takes a different path than intended due to some error.
- Specifications are correct and achievable.
- Processing bugs are only in control flow statements
- Data definition & access are correct

Observations

- Structured programming languages need less of path testing.
- Assembly language, Cobol, Fortran, Basic & similar languages make path testing necessary.

Control Flow Graph

A simplified, abstract, and graphical representation of a program's control structure using process blocks, decisions and junctions.

Process Block

Decisions

Junctions

Case Statement

Control Flow Graph Elements

Control Flow Graph Elements:

Process Block:

- A sequence of program statements uninterrupted by decisions or junctions with a single entry and single exit.

Junction:

- A point in the program where control flow can merge (into a node of the graph)
- Examples: target of GOTO, Jump, Continue

Decisions:

- A program point at which the control flow can diverge *(based on evaluation of a condition)*.
- Examples: IF stmt. Conditional branch and Jump instruction.

Case Statements:

- A Multi-way branch or decision.
- Examples: In assembly language: jump addresses table, Multiple GOTOs, Case/Switch

- For test design, Case statement and decision are similar.

Control Flow Graph Vs Flow Charts

Control Flow Graph	Flow Chart
Compact representation of the program	Usually a multi-page description
Focuses on Inputs, Outputs, and the control flow into and out of the block.	Focuses on the process steps inside
Inside details of a process block are not shown	Every part of the process block are drawn

Creation of Control Flow Graph from a program

- One statement to one element translation to get a Classical Flow chart
- Add additional labels as needed
- Merge process steps
- A process box is implied on every junction and decision
- Remove External Labels
- Represent the contents of elements by numbers.
- We have now **Nodes** and **Links**

```
        INPUT X, Y
        Z := X + Y
        V := X - Y
        IF Z >= 0  GOTO SAM
JOE:    Z := Z + V
SAM:    Z := Z - V
        FOR N = 0 TO V
        Z := Z - 1
        NEXT N
        END
```

One to One Flow Chart

Creation of Control Flow Graph from a program

- One statement to one element translation to get a Classical Flow chart
- Add additional labels as needed
- Merge process steps
- A process box is implied on every junction and decision
- Remove External Labels
- Represent the contents of elements by numbers.
- We have now **Nodes** and **Links**

```
        INPUT X, Y
        Z := X + Y
        V := X - Y
        IF Z >= 0  GOTO SAM
JOE:    Z := Z + V
SAM:    Z := Z - V
        FOR N = 0 TO V
        Z := Z - 1
        NEXT N
        END
```

Simplified Flow Graph

Creation of Control Flow Graph from a program

- One statement to one element translation to get a Classical Flow chart
- Add additional labels as needed
- Merge process steps
- A process box is implied on every junction and decision
- Remove External Labels
- Represent the contents of elements by numbers.
- We have now **Nodes** and **Links**

```
        INPUT X, Y
        Z := X + Y
        V := X - Y
        IF Z >= 0  GOTO SAM
JOE:    Z := Z + V
SAM:    Z := Z - V
        FOR N = 0 TO V
        Z := Z - 1
        NEXT N
        END
```

Simplified Flow Graph

Linked List Notation of a Control Flow Graph

Node	Processing, label, Decision	Next-Node
1	(BEGIN; INPUT X, Y; Z := X+Y ; V := X-Y)	: 2
2	(Z >= 0 ?)	: 4 (TRUE)
		: 3 (FALSE)
3	(JOE: Z := Z + V)	: 4
4	(SAM: Z := Z – V; N := 0)	: 5
5	(LOOP; Z := Z -1)	: 6
6	(N = V ?)	: 7 (FALSE)
		: END (TRUE)
7	(N := N + 1)	: 5

Path Testing Concepts

1. **Path** is a sequence of statements starting at an entry, junction or decision and ending at another, or possibly the same junction or decision or an exit point.

   Link is a single process (*block*) in between two nodes.

   Node is a junction or decision.

   Segment is a sequence of links. A path consists of many segments.

   Path segment is a succession of consecutive links that belongs to the same path. (3,4,5)

   Length of a path is measured by # of links in the path or # of nodes traversed.

   Name of a path is the set of the names of the nodes along the path. (1,2,3 4,5, 6)
   (1,2,3,4, 5,6,7, 5,6,7, 5,6)

   Path-Testing Path is an “entry to exit” path through a processing block.

2. Entry / Exit for a routines, process blocks and nodes.

Single entry and single exit routines are preferable.

Called well-formed routines.

Formal basis for testing exists.

Tools could generate test cases.

Control Flow Graphs and Path Testing

Multi-entry / Multi-exit routines: (ill-formed)

- A Weak approach: Hence, convert it to single-entry / single-exit routine.
- Integration issues:

  Large # of inter-process interfaces. Creates problem in Integration.
  More # test cases and also a formal treatment is more difficult.

- Theoretical and tools based issues
  - A good formal basis does not exist.
  - Tools may fail to generate important test cases.

Control Flow Graphs and Path Testing

Path Testing Concepts contd..

Convert a multi-entry / exit routine to a single entry / exit routine:

- Use an entry parameter and a case statement at the entry => single-entry

- Merge all exits to Single-exit point after setting one exit parameter to a value.

Path Testing Concepts contd..

Test Strategy for Multi-entry / exit routines

1. Get rid of them.
2. Control those you cannot get rid of.

3. Convert to single entry / exit routines.
4. Do unit testing by treating each entry/exit combination as if it were a completely different routine.

5. Recognize that integration testing is heavier
6. Understand the strategies & assumptions in the automatic test generators and confirm that they do (or do not) work for multi-entry/multi exit routines.

Path Testing Concepts

3. Fundamental Path Selection Criteria

A minimal set of paths to be able to do complete testing.

- Each pass through a routine from entry to exit, as one traces through it, is a **potential** path.
- The above includes the tracing of 1**..**n times tracing of an interactive block each separately.
- **Note**: A bug could make a mandatory path not executable or could create new paths not related to processing.

Complete Path Testing prescriptions:

1. Exercise every path from entry to exit.
2. Exercise every statement or instruction at least once.
3. Exercise every branch and case statement in each direction, at least once.

♦ Point 1 => point 2 and 3.
♦ Point 1 is impractical.
♦ Point 2 & 3 are not the same
♦ For a structured language, Point 3 => Point 2

Path Testing Concepts

Path Testing Criteria :

1. **Path Testing (P_∞):**

   Execute all possible control flow paths thru the program; but typically restricted to entry-exit paths.

   Implies 100% path coverage. Impossible to achieve.

2. **Statement Testing (P_1)** :

   Execute all statements in the program at least once under the some test.
   100% statement coverage => 100% node coverage.
   Denoted by **C1**
   C1 is a minimum testing requirement in the IEEE unit test standard: ANSI 87B.

3. **Branch Testing (P_2)** :

   Execute enough tests to assure that every branch alternative has been exercised at least once under some test.
   Denoted by **C2**
   Objective: 100% branch coverage and 100% Link coverage.

   For well structured software, branch testing & coverage include statement coverage

Control Flow Graphs and Path Testing

Picking enough (the fewest) paths for achieving C1+C2

1. Does every decision have Y & N (C2)?
2. Are call cases of case statement marked (C2)?
3. Is every three way branch covered (C2)?
4. Is every link covered at least once (C1)?

Make small changes in the path changing only 1 link or node at a time.

Revised path selection Rules

1. Pick the simplest and functionally sensible entry/exit path
2. Pick additional paths as small variations from previous paths. (pick those with no loops, shorter paths, simple and meaningful)
3. Pick additional paths but without an obvious functional meaning (only to achieve C1+C2 coverage).
4. Be comfortable with the chosen paths. play hunches, use intuition to achieve C1+C2
5. Don’t follow rules slavishly – except for coverage

4. Testing of Paths involving loops

Bugs in iterative statements apparently are not discovered by C1+C2.
But by testing at the boundaries of loop variable.

Types of Iterative statements

1. **Single loop statement.**
2. **Nested loops.**
3. **Concatenated Loops.**
4. **Horrible Loops**

Let us denote the Minimum # of iterations by n_{min}
the Maximum # of iterations by n_{max}
the value of loop control variable by V
the #of test cases by T
the # of iterations carried out by n

- Later, we analyze the Loop-Testing times

Testing of path involving loops…

1. Testing a Single Loop Statement (three cases)

Case 1. $n_{min} = 0$, $n_{max} = N$, no excluded values

1. Bypass the loop.
   If you can't, there is a bug, $n_{min} \neq 0$ or a wrong case.
2. Could the value of loop (control) variable V be negative?
   could it appear to specify a –ve n ?
3. Try one pass through the loop statement: n = 1
4. Try two passes through the loop statement: n = 2
   To detect initialization data flow anomalies:
   Variable defined & not used in the loop, or
   Initialized in the loop & used outside the loop.
5. Try n = typical number of iterations : $n_{min} < n < n_{max}$
6. Try n = n_{max} -1

7. Try n = n_{max}
8. Try n = n_{max} + 1.
   What prevents V (& n) from having this value?
   What happens if it is forced?

Control Flow Graphs and Path Testing

Testing of path involving loops…

Case 2. n_{min} = +ve, n_{max} = N, no excluded values

1. Try n_{min} - 1
   Could the value of loop (control) variable V be < n_{min}?
   What prevents that ?
2. Try n_{min}
3. Try n_{min} + 1

4. Once, unless covered by a previous test.
5. Twice, unless covered by a previous test.

4. Try n = typical number of iterations : n_{min} < n < n_{max}
5. Try n = n_{max} -1

6. Try n = n_{max}
7. Try n = n_{max} + 1.
   What prevents V (& n) from having this value?
   What happens if it is forced?

Note: only a case of no iterations, n = 0 is not there.

Case 3. Single loop with excluded values

1. Treat this as single loops with excluded values as two sets.

2. Example:

V = 1 to 20 excluding 7,8,9 and10

Test cases to attempt are for:

V = 0,1,2,4,6,7 and V = 10,11,15,19,20,21

(underlined cased are not supposed to work)

Testing of path involving loops…

2. Testing a Nested Loop Statement

- Multiplying # of tests for each nested loop => very large # of tests
- A test selection technique:

  1. Start at the inner-most loop. Set all outer-loops to Min iteration parameter values: **V_{min}**.
  2. Test the **V_{min}, V_{min} + 1, typical V, V_{max} - 1, V_{max}** for the inner-most loop. Hold the outer-loops to **V_{min}**. Expand tests are required for out-of-range & excluded values.

  3. If you have done with outer most loop, Go To step 5. Else, move out one loop and do step 2 with all other loops set to **typical values**.
  4. Do the five cases for all loops in the nest simultaneously.

- Assignment: check # test cases = 12 for nesting = 2, 16 for 3, 19 for 4.

Control Flow Graphs and Path Testing

Testing of path involving loops…

- Compromise on # test cases for processing time.
- Expand tests for solving potential problems associated with initialization of variables and with excluded combinations and ranges.
- Apply Huang's twice thorough theorem to catch data initialization problems.

Testing of path involving loops…

3. Testing Concatenated Loop Statements

- Two loops are concatenated if it's possible to reach one after exiting the other while still on the path from entrance to exit.
- If these are independent of each other, treat them as independent loops.
- If their iteration values are inter-dependent & these are same path, treat these like a nested loop.
- Processing times are additive.

Testing of path involving loops…

4. Testing Horrible Loops

- Avoid these.
- Even after applying some techniques of paths, resulting test cases not definitive.
- Too many test cases.
- Thinking required to check end points etc. is unique for each program.
- Jumps in & out of loops and intersecting loops etc, makes test case selection an ugly task.
- etc. etc.

Loop Testing Times

- Longer testing time for all loops if all the extreme cases are to be tested.
- Unreasonably long test execution times indicate bugs in the s/w or specs.

  Case: Testing nested loops with combination of extreme values leads to long test times.

  - Show that it's due to incorrect specs and fix the specs.
  - Prove that combined extreme cases cannot occur in the real world. Cut-off those tests.

  - Put in limits and checks to prevent the combined extreme cases.
  - Test with the extreme-value combinations, but use different numbers.

- The test time problem is solved by rescaling the test limit values.

  - Can be achieved through a separate compile, by patching, by setting parameter values etc..

Effectiveness of Path Testing

- Path testing (with mainly P1 & P2) catches ~65% of Unit Test Bugs ie., ~35% of all bugs.
- More effective for unstructured than structured software.

- **Limitations**
  - Path testing may not do expected coverage if bugs occur.
  - Path testing may not reveal totally wrong or missing functions.
  - Unit-level path testing may not catch interface errors among routines.
  - Data base and data flow errors may not be caught.
  - Unit-level path testing cannot reveal bugs in a routine due to another.
  - Not all initialization errors are caught by path testing.
  - Specification errors cannot be caught.

Effectiveness of Path Testing

- **A lot of work**
  - Creating flow graph, selecting paths for coverage, finding input data values to force these paths, setting up loop cases & combinations.
- Careful, systematic, test design will catch as many bugs as the act of testing.
  Test design process at all levels at least as effective at catching bugs as is running the test designed by that process.
- More complicated path testing techniques than P1 & P2
  - Between P_2 & P_α
    - Complicated & impractical
  - Weaker than P1 or P2.
    - For regression (incremental) testing, it's cost-effective

Control Flow Graphs and Path Testing

Predicates, Predicate Expressions

Path

- A sequence of process links (& nodes)

Predicate

- The logical function evaluated at a decision : True or False. *(Binary , boolean)*

Compound Predicate

- Two or more predicates combined with AND, OR etc.

Path Predicate

- Every path corresponds to a succession of True/False values for the predicates traversed on that path.
- A predicate associated with a path.
  " X > 0 is True " AND "W is either negative or equal to 122" is True
- Multi-valued Logic / Multi-way branching

Predicates, Predicate Expressions…

Predicate Interpretation

- The symbolic substitution of operations along the path in order to express the predicate solely in terms of the input vector is called **predicate interpretation**.

  An input vector is a set of inputs to a routine arranged as a one dimensional array.

- Example:

```
        INPUT X, Y
        ON X GOTO A, B, C
    A:  Z := 7  @  GOTO H
    B:  Z := -7  @ GOTO H
    C:  Z := 0  @ GOTO H

    H:  DO SOMETHING

    K:  IF X + Z > 0  GOTO  GOOD  ELSE  GOTO  BETTER
```

```
INPUT X
IF X < 0 THEN Y:= 2
ELSE  Y := 1
IF X + Y*Y > 0 THEN …
```

- Predicate interpretation may or may not depend on the path. IF -7 > 3 ..

- Path predicates are the specific form of the predicates of the decisions along the selected path after interpretation.

Predicates, Predicate Expressions…

Process Dependency

- An **input variable** is **independent** of the processing if its value does not change as a result of processing.
- An **input variable** is **process dependent** if its value changes as a result of processing.
- A **predicate** is **process dependent** if its truth value can change as a result of processing.
- A **predicate** is **process independent** if its truth value does not change as a result of processing.
- Process dependence of a predicate doesn't follow from process dependence of variables
- Examples:

	X + Y = 10	Increment X & Decrement Y.
	X is odd	Add an even # to X

- If all the input variables (on which a predicate is based) are process independent, then **predicate is process independent**.

Predicates, Predicate Expressions…

Correlation

- Two **input variables** are **correlated** if every combination of their values cannot be specified independently.
- **Variables** whose values can be specified independently without restriction are **uncorrelated**.
- A pair of predicates whose outcomes depend on one or more variables in common are **correlated predicates**.
- Every path through a routine is **achievable** only if all predicates in that routine are **uncorrelated**.
- If a routine has a loop, then at least one decision's predicate must be process dependent. Otherwise, there is an input value which the routine loops indefinitely.

Predicates, Predicate Expressions…

Path Predicate Expression

- Every selected path leads to an associated boolean expression, called the **path predicate expression**, which characterizes the input values (if any) that will cause that path to be traversed.
- Select an entry/exit path. Write down un-interpreted predicates for the decisions along the path. If there are iterations, note also the value of loop-control variable for that pass. Converting these into predicates that contain only input variables, we get a set of boolean expressions called path predicate expression.
- Example (inputs being numerical values):

If $X_5 > 0$ .OR. $X_6 < 0$ then

$X_1 + 3X_2 + 17 >= 0$
$X_3 = 17$
$X_4 – X_1 >= 14 X_2$

Predicates, Predicate Expressions…

A: $X_5 > 0$

B: $X_1 + 3X_2 + 17 >= 0$

C: $X_3 = 17$

D: $X_4 – X_1 >= 14 X_2$

E: $X_6 < 0$

F: $X_1 + 3X_2 + 17 >= 0$

G: $X_3 = 17$

H: $X_4 – X_1 >= 14 X_2$

Converting into the predicate expression form:

$$A\,B\,C\,D + E\,B\,C\,D \Rightarrow (A + E)\,B\,C\,D$$

If we take the alternative path for the expression: D then

$$(A + E)\,B\,C\,\overline{D}$$

Predicates, Predicate Expressions…

Predicate Coverage:

- Look at **examples** & possibility of bugs: A B C D A + B + C + D
  - Due to semantics of the evaluation of logic expressions in the languages, the entire expression may not be always evaluated.
  - A bug may not be detected.
  - A wrong path may be taken if there is a bug.
- Realize that on our achieving C2, the program could still hide some control flow bugs.
- **Predicate coverage:**
  - If all possible combinations of truth values corresponding to selected path have been explored under some test, we say predicate coverage has been achieved.
  - Stronger than branch coverage.
  - If all possible combinations of all predicates under all interpretations are covered, we have the equivalent of total path testing.

Testing blindness

- **coming to the right path** – even thru a wrong decision (at a predicate). Due to the interaction of some statements makes a buggy predicate work, and the bug is not detected by the selected input values.

- **calculating wrong number of tests** at a predicate by ignoring the # of paths to arrive at it.

- **Cannot be detected by path testing and need other strategies**

Testing blindness

- **Assignment blinding**: A buggy Predicate seems to work correctly as the specific value chosen in an assignment statement works with both the correct & buggy predicate.

Correct	Buggy	
X := 7	X := 7	
IF Y > 0 THEN …	IF X + Y > 0 THEN …	**(check for Y=1)**

- **Equality blinding**:
  - When the path selected by a prior predicate results in a value that works both for the correct & buggy predicate.

Correct	Buggy	
IF Y = 2 THEN …	IF Y = 2 THEN ..	
IF X + Y > 3 THEN …	IF X > 1 THEN …	**(check for any X>1)**

- **Self-blinding**
  - When a buggy predicate is a multiple of the correct one and the result is indistinguishable along that path.

Correct	Buggy	
X := A	X := A	
IF X - 1 > 0 THEN …	IF X + A -2 > 0 THEN …	**(check for any X,A)**

Achievable Paths

1. Objective is to select & test just enough paths to achieve a satisfactory notion of test completeness such as C1 + C2.

2. Extract the program's control flow graph & select a set of tentative covering paths.

3. For a path in that set, interpret the predicates.

4. Trace the path through, multiplying the individual compound predicates to achieve a boolean expression. Example: (A + BC) (D + E)

5. Multiply & obtain **sum-of-products** form of the **path predicate expression**:
   AD + AE + BCD + BCE

6. Each product term denotes a set of inequalities that, if solved, will yield an input vector that will drive the routine along the selected path.

7. A set of input values for that path is found when any of the inequality sets is solved.

   A solution found => **path is achievable**. Otherwise the path is **unachievable**.

Path Sensitization

It's the act of **finding a set of solutions to the path predicate expression**.

In practice, for a selected path finding the required input vector is not difficult. If there is difficulty, it may be due to some bugs.

Heuristic procedures:

Choose an easily sensitizable path set, & pick hard-to-sensitize paths to achieve more coverage.

Identify all the variables that affect the decisions. For process dependent variables, express the nature of the process dependency as an equation, function, or whatever is convenient and clear. For correlated variables, express the logical, arithmetic, or functional relation defining the correlation.

1. Identify correlated predicates and document the nature of the correlation as for variables. If the same predicate appears at more than one decision, the decisions are obviously correlated.
2. Start path selection with uncorrelated & independent predicates. If coverage is achieved, but the path had dependent predicates, something is wrong.

Path Sensitization… Heuristic procedures: contd..

4. If the coverage is not achieved yet with independent uncorrelated predicates, extend the path set by using correlated predicates; preferably process independent (not needing interpretation)

5. If the coverage is not achieved, extend the path set by using dependent predicates (typically required to cover loops), preferably uncorrelated.

6. Last, use correlated and dependent predicates.

7. For each of the path selected above, list the corresponding input variables. If the variable is independent, list its value. For dependent variables, interpret the predicate ie., list the relation. For correlated variables, state the nature of the correlation to other variables. Determine the mechanism (relation) to express the forbidden combinations of variable values, if any.

8. Each selected path yields a set of inequalities, which must be simultaneously satisfied to force the path.

Examples for Path Sensitization

1. Simple Independent Uncorrelated Predicates

2. Independent Correlated Predicates

3. Dependent Predicates

4. Generic

Examples for Path Sensitization..

1. Simple Independent Uncorrelated Predicates

4 predicates => 16 combinations
Set of possible paths = 8

Path	Predicate Values			
abcdef	A		$\overline{C}$	
aghcimkf	$\overline{A}$	B	C	D
aglmjef	$\overline{A}$	$\overline{B}$		$\overline{D}$

Path	Predicate Values			
abcdef	A		$\overline{C}$	
abcimjef	A		C	$\overline{D}$
abcimkf	A		C	D
aghcdef	$\overline{A}$	B	$\overline{C}$	
aglmkf	$\overline{A}$	$\overline{B}$		$\overline{D}$

A Simple case of solving inequalities. (obtained by the procedure for finding a covering set of paths)

2. Correlated Independent Predicates

Correlated paths => **some** paths are unachievable
ie., redundant paths.
ie., n decisions but # paths are < 2^n

Due to practice of saving code which makes the code very difficult to maintain.

Eliminate the correlated decisions.
Reproduce common code.

If a chosen sensible path is not achievable,

- there's a bug.
- design can be simplified.
- get better understanding of correlated decisions

Correlated decision removed & CFG simplified

3. Dependent Predicates

Usually most of the processing does not affect the control flow.

Use computer simulation for sensitization in a simplified way.

Dependent predicates contain iterative loop statements usually.

For Loop statements:

Determine the value of loop control variable for a certain # of iterations, and then work backward to determine the value of input variables (input vector).

4. The General Case

No simple procedure to solve for values of input vector for a selected path.

1. Select cases to provide coverage on the basis of **functionally sensible paths**.

Well structured routines allow easy sensitization.

Intractable paths may have a bug.

2. Tackle the path with the fewest decisions first. Select paths with least # of loops.

3. Start at the end of the path and list the predicates while tracing the path in **reverse**.

Each predicate imposes restrictions on the subsequent (in reverse order) predicate.

4. Continue tracing along the path. Pick the broadest range of values for variables affected and consistent with values that were so far determined.

5. Continue until the entrance & therefore have established a set of input conditions for the path.

If the solution is not found, path is not achievable, *it could be a bug.*

4. The General Case contd..

Alternately:

1. In the **forward** direction, list the decisions to be traversed.
   For each decision list the broadest range of input values.

2. Pick a path & adjust all input values. These restricted values are used for next decision.

3. Continue. Some decisions may be dependent on and/or correlated with earlier ones.

4. The path is unachievable if the input values become contradictory, or, impossible.

   If the path is achieved, try a new path for additional coverage.

Advantages & Disadvantages of the two approaches:

The forward method is usually less work.

you do not know where you are going as you are tracing the graph.

PATH INSTRUMENTATION

Output of a test:

Results observed. But, there may not be any expected output for a test.

Outcome:

Any change or the lack of change at the output.

Expected Outcome:

Any **expected** change or the lack of change at the output (predicted as part of design).

Actual Outcome:

Observed outcome

PATH INSTRUMENTATION

Coincidental Correctness:

When expected & actual outcomes match,

- Necessary conditions for test to pass are met.
- Conditions met are probably not sufficient.
  (the expected outcome may be achieved due to a wrong reason)

Path Instrumentation is what we have to do confirm that the outcome was achieved by the intended path.

PATH INSTRUMENTATION METHODS

1. General strategy:

1. Based on Interpretive tracing & use interpreting trace program.
2. A trace confirms the expected outcome is or isn't obtained along the intended path.
3. Computer trace may be too massive. Hand tracing may be simpler.

2. Traversal or Link Makers:

Simple and effective

1. Name every link.
2. Instrument the links so that the link is recorded when it is executed (during the test)
3. The succession of letters from a routine's entry to exit corresponds to the pathname.

Single Link Marker Instrumentation: An example

Sample path: i j n

Single Link Marker Instrumentation: Not good enough

Problem:

Processing in the links may be chewed open by bugs. Possibly due to GOTO statements, control takes a different path, yet resulting in the intended path again.

Double Link Marker Instrumentation:

The problem is solved.

Two link markers specify the path name and both the beginning & end of the link.

3. Link Counters Technique:

- Less disruptive and less informative.
- Increment a link counter each time a link is traversed. **Path length could confirm the intended path**.
- For avoiding the same problem as with markers, use double link counters. Expect an even count = double the length.
- Now, put a link counter on every link. *(earlier it was only between decisions)* If there are no loops, the link counts are = 1.
- Sum the link counts over a series of tests, say, a covering set. Confirm the total link counts with the expected.
- Using double link counters avoids the same & earlier mentioned problem.

3. Link Counters Technique: contd..

Check list for the procedure:

- Do begin-link counter values equal the end-link counter values?
- Does the input-link count of every decision equal to the sum of the link counts of the output links from that decision?
- Do the sum of the input-link counts for a junction equal the output-link count for that junction?
- Do the total counts match the values you predicted when you designed the covering test set?

This procedure and the checklist could solve the problem of Instrumentation.

PATH INTRUMENTATION techniques...

Limitations

- Instrumentation probe (marker, counter) **may disturb the timing relations** & hide racing condition bugs.

- Instrumentation probe (marker, counter) **may not detect location dependent bugs**.

  If the presence or absence of probes modifies things (for example in the data base) in a faulty way, then the **probes hide the bug** in the program.

PATH INTRUMENTATION - IMPLEMENTATION

For Unit testing : Implementation may be provided by a comprehensive test tool.

For higher level testing or for testing an unsupported language:

- Introduction of probes could introduce bugs.
- Instrumentation is more important for higher level of program structure like transaction flow
- At higher levels, the discrepancies in the structure are more possible & overhead of instrumentation may be less.

For Languages supporting conditional assembly or **compilation**:

- Probes are written in the source code & tagged into categories.
- Counters & traversal markers can be implemented.
- Can selectively activate the desired probes.

For language not supporting conditional assembly / compilation:

- Use macros or function calls for each category of probes. This may have less bugs.
- A general purpose routine may be written.

In general:

- Plan instrumentation with probes in levels of increasing detail.

Implementation & Application of Path Testing

1. Integration, Coverage, and Paths in Called Components

- Mainly used in Unit testing, especially new software.

- **In an Idealistic bottom-up integration test process** – integrating one component at a time. Use stubs for lower level component (sub-routines), test interfaces and then replace stubs by real subroutines.

- **In reality,** integration proceeds in associated blocks of components. Stubs may be avoided. Need to think about paths inside the subroutine.

To achieve C1 or C2 coverage:

- Predicate interpretation may require us to treat a subroutine as an in-line-code.
- Sensitization becomes more difficult.
- Selected path may be unachievable as the called components' processing may block it.

Weaknesses of Path testing:

- It assumes that effective testing can be done one level at a time without bothering what happens at lower levels.
- predicate coverage problems & blinding.

Implementation & Application of Path Testing

2. Application of path testing to **New Code**

- Do Path Tests for C1 + C2 coverage
- Use the procedure similar to the idealistic bottom-up integration testing, using a mechanized test suite.
- A path blocked or not achievable could mean a bug.
- When a bug occurs the path may be blocked.

Implementation & Application of Path Testing

3. Application of path testing to **Maintenance**

- Path testing is applied first to the modified component.
- Use the procedure similar to the idealistic bottom-up integration testing, but without using stubs.
- Select paths to achieve C2 over the changed code.
- Newer and more effective strategies could emerge to provide coverage in maintenance phase.

Implementation & Application of Path Testing

4. Application of path testing to **Rehosting**

- Path testing with C1 + C2 coverage is a powerful tool for rehosting old software.
- Software is rehosted as it's no more cost effective to support the application environment.
- Use path testing in conjunction with automatic or semiautomatic structural test generators.

Application of path testing to **Rehosting..**

Process of path testing during rehosting

- A translator from the old to the new environment is created & tested. Rehosting process is to catch bugs in the translator software.
- A complete C1 + C2 coverage path test suite is created for the old software. Tests are run in the old environment. The outcomes become the specifications for the rehosted software.
- Another translator may be needed to adapt the tests & outcomes to the new environment.
- The cost of the process is high, but it avoids risks associated with rewriting the code.
- Once it runs on new environment, it can be optimized or enhanced for new functionalities (which were not possible in the old environment.)

UNIT-II

Transaction Flow Testing: Transaction flows, transaction flow testing techniques.

Dataflow testing -Basics of dataflow testing, strategies in dataflow testing, application of dataflow testing.

Definitions of Transaction,Transaction-flow, TFG

Transaction-flow

Transaction-flow represents a system's processing. Functional testing methods are applied for testing T-F.

Transaction-flow Graph

TFG represents a behavioral (functional) model of the program (system) used for functional testing *by an independent system tester*.

Transaction

- is a unit of work seen from system's user point of view.
- consists of a sequence of operations performed by a system, persons or external devices.
- is created (birth) due to an external act & up on its completion (closure), it remains in the form of historical records.

A Simple Transaction

Example: the sequence of steps in a transaction in an online information retrieval system

1. accept inputs
2. Validate inputs (Birth of tr.)
3. Transmit ack. to the user
4. Process input
5. Search file
6. Request direction from user
7. Accept Inputs
8. Validate inputs
9. Process the request
10. Update file
11. Transmit output
12. Record transaction in log & cleanup (Closure)

Users View of a transaction : Single step

Systems view : Sequence of many operations

Example of a Transaction flow (diagram)

User (terminal) ↔ Terminal controller ↔ CPU

Definitions

Transaction-flow Graph : a scenario between users & computer

Transaction-flow : an internal sequence of events in processing a transaction

Uses of Transaction-flow

Specifying requirements of big, online and complicated systems.

Airline reservation systems, air-traffic control systems.

Loops are less as compared to CFG. Loops are used for user input error processing

Implementation of Transaction-Flow (in a system)

- Implicit in the design of system's control structure & associated database.
- No direct one-to-one correspondence between the "**processes**" and "**decisions**" of transaction-flow, and the corresponding program **component**.
- A **transaction-flow** is a path taken by the transaction through a succession of processing modules.
- A transaction is represented by a **token**.
- A **transaction-flow graph** is a pictorial representation of what happens to the tokens.

S : Scheduler **A, B, C, D, E : Processes**

Implementation of Transaction-Flow

System Control Structure (architecture of the implementation) :

Executive / Dispatcher Flowchart *(a sample sequence)*

System control structure

System is controlled by a scheduler …

A Transaction is created by filling in a **Transaction Control Block** (TCB) by user inputs and by placing that **token** on input Q of Scheduler.

Scheduler examines and places it on appropriate process Q such as **A**. When **A** finishes with the Token, it places the TCB back on the scheduler Q.

Scheduler routes it to the next process after examining the token :

1. It contains tables or code to route a token to the next process.
2. It may contain routing information only in tables.
3. Scheduler contains no code / data. Processing modules contain code for routing.

Implementation of Transaction-Flow

Transaction Processing System (simplified):

- There are many Tr. & Tr-flows in the system.
- Scheduler invokes processes A to E as well as disk & tape read & writes.
- The order of execution depends on priority & other reasons.

Cyclic structure like in this example is common in process control & communication systems.

- The criteria for implementation mechanism depends on performance and resource optimization.

A perspective of Transaction-Flow

Transaction-flow testing is a block box technique. *(as we assumed nothing regarding computer, communications, environment, O.S., transaction identity or structure or state.)*

1. TFG is a kind of DFG.
   - TFG has tokens, & DFG has data objects with history of operations applied on them.
   - Many techniques of CFG apply to TFG & DFG
2. **Decision nodes** of TFG have exception exits to the central recovery process.
3. So, we ignore the **effect of interrupts** in a Transaction-flow.

Splits of transactions (Births)

1. A decision point in TFG

2. Biosis

3.Mitosis

Mergers of transactions

1. Junction

2. Absorption

3. Conjugation

TFG – splitting, merging Transactions

NOTES:

- Multiple meanings now for decision and junction symbols in a TFG.
- Simple TFG model is **not enough** to represent multi-processor systems & associated coordination systems.
- **Petrinet model** uses operations for all the above. But Petrinets are applied to H/W, N/W protocol testing – but not **Software**.

Simplify TFG model

- **Add New Tr-Flows for** Biosis, Mitosis, Absorption, Conjugation
- **Problems for programmer, designer & test designer.**
- Need to **design specific tests** – possibility of bugs.

Transaction-flow Structure

Reasons for Unstructuredness

1. Processes involve Human Users

1. Part of Flow from External Systems

1. Errors, Failures, Malfunctions & Recovery Actions

1. Transaction Count, Complexity. Customer & Environment

Transaction-flow Structure

Reasons for Unstructuredness …

5. **New Transactions, & Modifications**

6. **Approximation to Reality**

   - Attempt to Structure

Transaction - Flow Testing - Steps

First, Build / Obtain Transaction Flows

- **Represent Explicitly**
- **Design details the Main Tr-Flows**
- **Create From PDL**
- **HIPO charts & Petrinet Representations**

Objective – Trace the transaction

Transaction - Flow Testing - Steps

1. **Inspections, Reviews & Walkthroughs**

   Start From Preliminary Design

   1. **Conducting Walkthroughs**

      - **Discuss enough Transaction Types (**98% Transactions**)**
      - **User needs & Functional terms** *(Design independent)*
      - **Traceability to Requirements**

Transaction - Flow Testing - Steps

1. **Inspections, Reviews & Walkthroughs …**

2. **Design Tests for C1 + C2 coverage**

3. **Additional Coverage (> C1+C2)**

   - Paths with loops, extreme values, domain boundaries
   - **Weird cases, long & potentially troublesome Tr.**

4. **Design Test cases for Tr. Splits & mergers**

5. **Publish Selected Test Paths early**

6. **Buyer's Acceptance – functional & acceptance tests**

Transaction - Flow Testing Techniques

2. Path Selection

1. **Covering Set (C1+C2) of Functionally Sensible Tr.**

2. **Add Difficult Paths**

   - Review with designers & implementers
   - Exposure of interface problems & duplicated processing
   - Very few Implementation bugs may remain

Transaction-flow Path Covering Set belongs in System Feature Tests

Transaction - Flow Testing Techniques

3. Sensitization

1. Functionally Sensible Paths – Simple
2. Error, Exception, External Protocol Interface Paths - Difficult

Testing Tr.–Flows with External Interfaces

- Use patches & break points, mistune, and break the rules,

Transaction - Flow Testing Techniques

4. Instrumentation

1. **Link Counters are not Useful.**
2. **Need**
   - Trace
   - Queues on which Tokens resided
   - Entries to & Exits from Dispatcher
   - A Running Log

Make Instrumentation as part of System Design

Transaction - Flow Testing Techniques

5. Test Data bases

1. **Design & Maintenance of a Test Data base - Effort**
2. Mistakes
   - Unawareness about design of a centrally administered test DB
   - Test DB design by Testers
   - Using one DB for all tests (need 4 to 5)

Need experienced System & Test Designers

Transaction - Flow Testing Techniques

6. Test Execution

1. **Use Test Execution Automation**

2. Have to do a large # of Tests for C1+C2 coverage

Transaction - Flow Testing - Implementation

1. **Transaction based systems**
   - **TCB**

1. **Centralized, Common Processing Queues**
   - **Just O(n) Queues for Links of $O(n^2)$**

2. **Transaction Dispatcher**
   - **Uses tables & Finite State Machines**

3. **Recovery & Other Logs**
   - **Key events in Tr – Flow**

4. **Self-Test Support**
   - **Privileged modes in Transaction control tables**

Anomaly

Unreasonable processing on data

- Use of data object before it is defined
- Defined data object is not used

- Data Flow Testing (DFT) uses Control Flow Graph (CFG) to explore dataflow anomalies.

  - DFT Leads to testing strategies between P_{∞} and P1 / P2

Data - Flow Testing - Basics

Definition:

DFT is a family of test strategies based on selecting paths through the program's control flow in order to explore the sequence of events related to the status of data objects.

Example:

Pick enough paths to assure that every data item has been initialized prior to its use, or that all objects have been used for something.

Motivation

- **Confidence in the program**
- **Data dominated design.** *Code migrates to data..*
- **Source Code for Data Declarations**
- **Data flow Machines vs Von Neumann's**
    - **Abstract M I M D**
    - **Language & compiler take care of parallel computations**

Program Control flow with Von Neumann's paradigm

Given m, n, p, q, find e.

e = (m+n+p+q) * (m+n-p-q)

a := m + n
b := p + q
c := a + b
d := a - b

e := c * d

a = n+m
↓
b=p+q
↓
c=a+b
↓
d=a-b
↓
e=c*d

Multiple representations of control flow graphs possible.

Program Flow using Data Flow Machines paradigm

```
BEGIN
 PAR DO
        READ m, n, n, p, q
END PAR
PAR DO
        a := m+n
        b := p+q
END PAR
PAR DO
        c := a+b
        d := a-b
END PAR
PAR DO
        e := c * d
END PAR
END
```

The interrelations among the data items remain same.

Data - Flow Testing - Basics - Motivation

- **Control flow graph**
  - **Multiple representations**

- **Data Flow Graph**

A spec. for relations among the data objects.

Covering DFG => Explore all relations under some test.

Assumptions

- **Problems in a control flow**
- **Problems with data objects**

Data Flow Graphs (DFG)

- It is a graph with nodes & directed links
- Test the Von Neumann way -

  Convert to a CFG

  Annotate : program actions (weights)

Data Object State & Usage

Program Actions (d, k, u):

Defined (created)	-	explicitly or implicitly	(d)
Killed (released)	-	directly or indirectly	(k)
Used	-	(u)	

- In a calculation - (c)
- In a predicate - directly or indirectly (p)

Data Flow Anomalies

A Two letter sequence of Actions (d, k, u)

dd	:	harmless, suspicious
dk	:	probably a bug.
du	:	normal
kd	:	normal
kk	:	harmless, but probably a bug
ku	:	a bug
ud	:	normal. Redefinition.
uk	:	normal
uu	:	normal

Program Flow using Data Flow Machines paradigm

```
BEGIN
 PAR DO
        READ m, n, n, p, q
END PAR
PAR DO
        a := m+n
        b := p+q
END PAR
PAR DO
        c := a+b
        d := a-b
END PAR
PAR DO
        e := c * d
END PAR
END
```

The interrelations among the data items remain same.

Actions on data objects

-	no action from START to this point From this point till the EXIT
- d	normal
- u	anomaly
- k	anomaly
k-	normal
u -	normal - possibly an anomaly
d -	possibly anomalous

Data Flow Anomaly State graph

- **Data Object State**
  - **K, D, U, A**
- **Processing Step**
  - **k, d, u**

Data Flow Anomaly State graph

- **Object state**
- **Unforgiving Data flow state graph**

Data Flow Anomaly State graph

Forgiving Data flow state graph

A ⇒ DD, DK, KU

Data Flow State Graphs

- **Differ in processing of anomalies**
- **Choice depends on Application, language, context**

Static vs Dynamic Anomaly Detection

- Static analysis of data flows
- Dynamic analysis

  Intermediate data values

Insufficiency of Static Analysis (for Data flow)

1. Validation of Dead Variables
2. Validation of pointers in Arrays
3. Validation of pointers for Records & pointers
1. Dynamic addresses for dynamic subroutine calls
2. Identifying False anomaly on an unachievable path
1. Recoverable anomalies & Alternate state graph
2. Concurrency, Interrupts, System Issues

Data Flow Model

- Based on CFG
- CFG annotated with program actions
- link weights : dk, dp, du etc..
- Not same as DFG
- For each variable and data object

Procedure to Build:

1. Entry & Exit nodes
1. Unique node identification
1. Weights on out link
2. Predicated nodes
3. Sequence of links
    1. Join
    2. Concatenate weights
    3. The converse

Example:

$$Z = b + \frac{a^n - 1}{a - 1}$$

```
START
INPUT a, b, n
Z := 0
IF   a = 1  THEN    Z  :=  1
GOTO  DONE1
r := 1    c := 1
POWER:
    c :=  c * a
    r  :=  r  + 1
    IF   r <= n   THEN     GO TO POWER
    Z := (c – 1) / (a – 1)
DONE1:
Z := b + Z

END
```

CFG for the Example

Data - Flow Testing - Basics – Data Flow model

CFG annotated – Data Flow Model for Z

Data - Flow Testing - Basics – Data Flow model

CFG annotated – Data Flow Model for c

Data - Flow Testing - Basics – Data Flow model

CFG annotated – Data Flow Model for r

Data - Flow Testing - Basics – Data Flow model

CFG annotated – Data Flow Model for b

Data - Flow Testing - Basics – Data Flow model

CFG annotated – Data Flow Model for n

Data - Flow Testing - Basics – Data Flow model

CFG annotated – Data Flow Model for a

Data - Flow Testing - Basics – Data Flow model

A DFM for each variable

Single DFM for multiple variables

Use weights subscripted with variables

Data - Flow Testing – Data Flow Testing Strategies

- A structural testing strategy (**path testing**)
- Add, data flow strategies **with link weights**
- Test path segments to have a **'d'** (or **u, k, du, dk**)

Data - Flow Testing – Data Flow Testing Strategies

DEFINITIONS

- w.r.t. a variable or data object ‘v’
- Assume all DF paths are achievable

1. Definition-clear path segment
   no k, kd

2. Loop-free path segment

2. Simple path segment

3. du path from node i to k

   - definition-clear & simple c
   - definition-clear & loop-free p

DFT Strategies

1. All-du paths (ADUP)
2. All uses (AU) strategy
3. All p-uses/some c-uses and All c-uses/some p-uses
1. All Definitions Strategy
1. All p-uses, All c-uses Strategy

Purpose:

Test Design, Develop Test Cases

Data - Flow Testing – Data Flow Testing Strategies

1. All-du paths (ADUP)

- Strongest DFT
- Every du path for every variable for every definition to every use

2. All uses (AU) strategy

- At least one definition clear path segment from every definition of every variable to every use of that definition be exercised under some test.
- At least one path segment from every definition to every use that can be reached from that definition.

3. All p-uses/some c-uses and All c-uses/some p-uses

- APU + c
  - Stronger than P2
- ACU + p
  - Weaker than P2

4. All Definitions Strategy (AD)

- Cover every definition by **at least one** p or c
- Weaker than ACU + p and APU + c

5. All-Predicate Uses, All-Computational Uses Strategy

- **APU :**
  - **Include definition-free path for every definition of every variable from the definition to predicate use.**
- **ACU :**
  - **Include for every definition of every variable include at least one definition-free path from the definition to every computational use.**

Ordering the strategies

Testing, Maintenance & Debugging in the Data Flow context

Slicing:

- A static program **slice** is a part of a program defined wrt a variable '**v**' and a statement '**s**'; It is the set of all statements that could affect the value of '**v**' at stmt '**s**'.

```
Stmt1     var v
stmt2
Stmt3     var v
Stmt4     var v

Stmt s     var  v
```

Testing, Maintenance & Debugging in the Data Flow context

Dicing:

- A program dice is a part of slice in which all stmts. which are known to be correct have been removed.
- Obtained from ‘slice’ by incorporating correctness information from testing / debugging.

Testing, Maintenance & Debugging in the Data Flow context

Debugging:

- Select a slice.
- Narrow it to a dice.
- Refine the dice till it’s one faulty stmt.

Testing, Maintenance & Debugging in the Data Flow context

Dynamic Slicing:

- Refinement of static slicing
- Only achievable paths to the stmt ‘s’ in question are included.

Slicing methods bring together testing, maintenance & debugging.

Application of DFT

- **Comparison Random Testing, P2, AU - by Ntafos**
  - **AU detects more bugs than**
    - **P2 with more test cases**
    - **RT with less # of test cases**
- **Comparison of P2, AU - by Sneed**
  - **AU detects more bugs with 90% Data Coverage Requirement.**

Application of DFT

- **Comparison of # test cases for ACU, APU, AU & ADUP**
  - **by Weyuker using ASSET testing system**
  - **Test Cases Normalized. t = a + b * d** *d = # binary decisions*
  - **At most d+1 Test Cases for P2** *loop-free*
  - **# Test Cases / Decision**

ADUP > AU > APU > ACU > revised-APU

Application of DFT

Comparison of # test cases for ACU, APU, AU & ADUP by Shimeall & Levenson

Test Cases Normalized. t = a + b * d *(d = # binary decisions)*

At most d+1 Test Cases for P2 *loop-free*

 # Test Cases / Decision

ADUP ~ ½ APU*

AP ~ AC

Application of DFT

DFT vs P1, P2

- **DFT is Effective**
- **Effort for Covering Path Set ~ Same**
- **DFT Tracks the Coverage of Variables**
- **Test Design is similar**

DFT - TOOLS

- Cost-effective development
- Commercial tools :
- Can possibly do Better than Commercial Tools
  - Easier Integration into a Compiler
  - Efficient Testing

UNIT-III

Domain Testing: domains and paths, Nice and ugly domains, domain testing, domains and interfaces testing, domain and interface testing, domains and testability.

Domain Testing Model

Domain Testing

Views Programs as input data classifiers

Verifies if the classification is correct

Domain Testing Model

Two Views

- Based on Specs
  - Functional Testing
- Based on Implementation information
  - Structural Technique

Domain Testing Model

- Variables
  - Simple combinations of two variables
  - Numbers from $-\propto$ to $+\propto$
- Input variables as numbers
- Loop-free Programs

Domain Testing Model

- Structural Knowledge is not needed. Only Specs.
- For each Input Case,
  - A Hypothetical path for Functional testing
  - An Actual path for Structural testing

Domain (simplified)

- **A Single Connected Set of numbers**
- **No arbitrary discrete sets**
- **Defined by the boundaries**
  - One or more boundaries
  - Specified by predicates
  - Bugs likely at the boundaries
- One or more variables

Predicates

- Interpretation
  - Structural Testing - CFG
  - Functional Testing - DFG
- Specifies the Domain Boundary
- n Predicates in Sequence $\Rightarrow$ 2^n domains
  - Or, just two domains

A .AND. B .AND. C

Paths

- At least One PATH through the Program
- More paths for disconnected domains

Summary:

- Domain for a loop-free program corresponds to a set of numbers defined over the input vector
- For every domain, there is at least one path thru the routine, along which that domain's processing is done
- The set of interpreted predicates traversed on that path (ie., the path's predicate expression) defines the domain's boundaries.

Domain Testing - Domains & Paths - Domain Closure

Boundary: Closed / Open

Domain Testing - Domains & Paths - Domain Closure

Boundary: Closed / Open

Domain Testing - Domains & Paths - Domain Dimensionality

- **One dimension per variable**
- **At least one predicate**
  - Slices thru previously defined Domain
- **Slicing Boundary**
- **N-spaces are cut by Hyperplanes**

Domain Testing - Domains & Paths – Bug Assumptions

- Processing is OK. Domain definition may be wrong.

  ⇒ Boundaries are wrong.

  ⇒ Predicates are wrong.

- Once input vector is set on the right path, it's correctly processed.

- More bugs causing domain errors …

Domain Testing - Domains & Paths – Bug Assumptions..

- **Double-zero representation**
- **Floating point zero check**
- **Contradictory domains**

- **Ambiguous domains**

Domain Testing - Domains & Paths – Bug Assumptions..

- **Over-specified domains**

X > 3 .AND. X < 2 .AND. Y > 3

- **Boundary errors**

Boundary closure

Shifted

Domain Testing - Domains & Paths - Bug Assumptions

- **Boundary errors….**

Tilted

Missing

Extra boundary

Domain Testing - Domains & Paths - Bug Assumptions

- **Closure Reversal**

- **Faulty Logic**

Simplification of compound predicates

Domain Testing - Domains & Paths - Bug Assumptions

Strategy for domain testing in 2-dim

Domain Testing - Domains & Paths - Restrictions

In General…

1. Coincidental Correctness

DT cannot detect
Example

- **Representative outcome**

Partition testing

Input Equivalence

Domain Testing Model

3. Simple boundaries & Compound predicates

X >=0 .AND. X <= 16

Y = 1 .AND. X >=0 .AND. X <= 16

Compound predicates….

- **Impact of .OR.**
  - Concave, Disconnected
  - Adjacent domains with same function
  - Example

A B C + D E F

- **Eliminate compound predicates**

4. Functional Homogeneity of Bugs

- **Functional form still Retained**

$$a x + b y >= c$$

- **Bugs are only in a, b, c**

5. Linear Vector Space

- Linear boundary predicate, Interpreted
- Simple relational operators

- Conversion to linear vector space
  - 2-d Polar co-ordinates
  - Polynomials

- Problems with Non-linear Boundaries

6. Loop-free Software

Predicate for each iteration

Loop over the entire transaction

Definite loop

Nice Domains

Requirements

- Bugs $\Rightarrow$ ill-defined domains

Before DT

- Analyze specs
- Make the Boundary Specs Consistent & Complete

Implemented Domains

- Complete, Consistent & Process all inputs

Specified Domains

- Incomplete, Inconsistent

Programmer's / Designer's Effort

Nice Domains

- Linear, Complete, Systematic, Orthogonal, Consistently Closed, & Convex

Nice Domains

Advantages

- **Ease of DT**
- **Fewer Bug Occurrences**

Nice Domains

Boundaries are

1. Linear
2. Complete
3. Systematic
4. Orthogonal
5. Closure consistency
6. Convex
7. Simply connected

1. Linear Boundaries

Interpreted linear inequalities

n-dim Hyperplane:

n+1 Points

n+1 + 1 Test Cases

Non-Linear

- **Transform**

2. Complete Boundaries

- Span the total number space (-∝, +∝)
  - One set of Tests

Incomplete…

- Reasons
- Tests

3. Systematic Boundaries

- Linear Inequalities differing by a constant

$f_j(X) \geq k_j$ or, $f_j(X) \geq g(j, c)$ $g(j, c) = j + k * c$

- Parallel lines
- Identical Sectors in a Circle

- **DT**
  - Test a domain $\Rightarrow$ Tests for other Domains

4. Orthogonal Boundaries

- Two boundaries or, boundary sets
- Parallel to axes

- **DT**
  - Each Set Independently
  - # Tests α O (n)

Orthogonal Boundaries

- Tilted sets
  - transformation
  - Test cases: O(n)

- Concentric circles with radial lines
  - Rectangular coordinates
  - Polar

    $r \geq a_j$ .AND. $r < a_{j+1}$ .AND.

    $\theta \geq \theta_j$ .AND. $\theta < \theta_{j+1}$

Non-Orthogonal Boundaries

- Test Intersections $\Rightarrow$ $O(n^2) + O(n)$

5. Closure Consistency

- A Simple pattern in all boundary closures
- Example
  - Same relational operator for systematic boundaries

6. Convex Domain

- Line joining any two points lies with in the domain

- DT
  - n on-points & 1 off-pt

Concave

- " But, However, Except, Or … " in Specs
- Handle with special care

7. Simply Connected Domain

- In a single piece

- 2 complementary domains
  - D1: Convex $\Rightarrow$ D2: Concave, not-Connected

- Programmer / Designer
  - Convex part First

Generally,

From Bad Specs

Programmer / Designer Simplifies => Ugly to Good

possibility of Introduction of more bugs ?!?

Causes

1. Non-linear Boundaries

1. Ambiguities & Contradictions

1. Simplifying the topology

1. Rectifying boundary closures

1. Non-linear Boundaries

- Transform

2. Ambiguities

- Holes in input vector space.
- Missing boundary
- Detected by Specification languages & tools.

2. Contradictions..

- Overlapping of
  - Domain Specs
  - Closure Specs

3. Simplifying the Topology….

- Complexity !
  - Concavity, Holes, Disconnectedness

3. Simplifying the Topology….

- Smoothing out concavity

- Filling in Holes

3. Simplifying the Topology….

- Joining the pieces

Correct:

- Connect disconnected boundary segments
- Extend boundaries to infinity

4. Rectifying Boundary Closures.

- make closures in one direction for parallel boundaries with closures in both directions
- Force a Bounding Hyperplane to belong to the Domain.

Consistent Direction

Inclusion / Exclusion Consistency

General DT Strategy

1. Select test points near the boundaries.
2. Define test strategy for each possible bug related to boundary
3. Test points for a domain useful to test its adjacent domain.
4. Run the tests. By post test analysis determine if any boundaries are faulty & if so how?
5. Run enough tests to verify every boundary of every domain

DT for Specific Domain Bugs

Generally,

- Interior point
- Exterior point
- Epsilon neighborhood
- Extreme point
- On point

DT for Specific Domain Bugs

Domain D1

Domain Testing - Domains & Paths – Domain Testing

1. 1-d Domains
2. 2-d Domains
3. Equality & inequality Predicates
4. Random Testing
5. Testing n-dimensional Domains

Testing 1-d Domains : Bugs with open boundaries

Closure Bug

Shift left Bug

Shift Right Bug

Testing 1-d Domains : Bugs with open boundaries

Missing Boundary

Extra Boundary

Bugs with Closed Boundaries : Similar to the above

2. Testing 2-d Domains

- Closure bug
- Boundary Shift : up / down
- Tilted Boundary
- Extra Boundary
- Missing Boundary

Domain Testing - Domains & Paths – Domain Testing

2 n : Domains share tests
3 n : no sharing of tests by domains

Strategy for domain testing in 2-dim

3. Equality & Inequality Predicates

- An Equality predicate defines a line in 2-d

4. Random Testing

- A Point in the center : verifies computation

5. Testing n-Dimensional Domains (strategy)

n-dimensions, p boundary segments

- (n+1)*p test cases : n on points & 1 off point
- Extreme pt shared : 2 * p points
- Equalities over m-dimensions create a subspace of n-m dimensions
- Orthogonal domains with consistent boundary closures, orthogonal to the axes & complete boundaries
    - Independent testing

Procedure for solving for values

Simple procedure. Need tools.

1. Identify input variables

2. Identify variables which appear in domain-defining predicates, such as control-flow predicates

Procedure

3. Interpret all domain predicates in terms of input variables:
   - Transform non-linear to linear
   - Find data flow path

4. Predicate expression with p # predicates.
   Find # domains : $< 2^p$

3. Solve inequalities for extreme points

4. Use extreme points to solve for nearby on points …

Domain Testing

Effectiveness

- Cost effective
- Bugs on boundaries, extreme points
- Hardware logic testing – tool intensive

Domains & Interface Testing

- Domain
- Range

Domain Testing

Domains & Interface Testing

- Span compatibility

Domain Testing

Interface Range/Domain Compatibility Testing

- Test each var independently
- Find an inverse function
- Build a super domain

Domain Testing

Domains and Testability

- **Linearizing transformations**
  - Polynomial
  - Rectangular to polar
  - Generic & rational => Taylor series

Domains and Testability

Perform transformations for better testing.

- **Co-ordinate transformations**

Domains and Testability

- Canonical Program Form

UNIT-IV

Paths, Path products and Regular expressions: Path products & path expression, reduction procedure, applications, regular expressions & flow anomaly detection.

Logic Based Testing: Overview, decision tables, path expressions, kv charts, specifications.

PURPOSE: APPLICATIONS

1. How many paths in a flow-graph?

   Maximum, minimum etc.

2. The probability of getting to a point in a program ?

   (to a node in a flow graph)

3. The mean processing time of a routine (a flow graph)

4. Effect of Routines involving complementary operations :

   (Push / Pop & Get / Return)

PURPOSE & APPLICATIONS

1. **Check for data flow anomalies.**

2. **Regular expressions are applied to problems in test design & debugging**

3. electronics engineers use flow graphs to design & analyze circuits & logic designers.

1. Software development, testing & debugging tools use flow graph analysis tools & techniques.

1. These are helpful for test tool builders.

Motivation

1. Flow graph is an abstract representation of a program.

2. A question on a program can be mapped on to an equivalent question on an appropriate flow graph.

3. It will be a foundation for syntax testing & state testing

Path Products & expressions - Definitions

Path Expression:

An algebraic representation of sets of paths in a flow graph.

Regular Expression:

Path expressions converted by using arithmetic laws & weights into an algebraic function.

Path Products & expressions – Path Product

- Annotate each link with a name.
- The pathname as you traverse a path (segment) expressed as concatenation of the link names is the **path product**.

- Examples of path products between 1 & 4 are:

a b d a b c b d a b c b c b d ….

Path Expression

Simply: **Derive using path products.**

Example:

{ a b d , a b c b d, a b c b c b d , ….. }

abd + abcbd + abcbcbd + ….

Path Products & expressions – Path Expression

Example:

{ abcd , abfhebcd , abfigebcd , abfijd }

abcd + abfhebcd + abfigebcd + abfijd

Path Products & expressions – Path Expression

Path name for two successive path segments is the concatenation of their path products.

X = abc	**Y = def**	**XY = abcdef**
a X = aabc	**X a = abca**	**XaX = abcaabc**
X = ab + cd	**Y = ef + gh**	**XY = abef + abgh + cdef + cdgh**

Path Products & expressions – Path Segments & Products

- **Loops:**

$a^1 = a$ $a^2 = aa$ $a^3 = aaa$ $a^n = aaaaa \ldots$ n times

$X = abc$ $X^1 = abc$ $X^2 = (abc)^2 = abcabc$

- **Identity element**

$a^0 = 1$ $X^0 = 1$ (path of length 0)

Path Product

- **Not Commutative:**

  $XY \neq YX$ in general

- **Associative**

  $A(BC) = (AB)C = ABC$: Rule 1

Path Products & expressions – Path Product

Denotes a set of paths in parallel between two nodes.

- Commutative

$$X + Y = Y + X$$: **Rule 2**

- Associative

$$(X + Y) + Z = X + (Y + Z) = X + Y + Z$$: **Rule 3**

- Distributive

$$A(B + C) = AB + AC$$
$$(A + B)C = AC + BC$$: **Rule 4**

Path Products & expressions – Path Products

- **Absorption**

$X + X = X$: **Rule 5**

X + any subset of X = X

$X = a + bc + abcd$ $X + a = X + bc + abcd = X$

Path Products & expressions – Path Products

- **Loop:**

An infinite set of parallel paths.

$$b^* = b^0 + b^1 + b^2 + b^3 + \ldots\ldots$$

$$X^* = X^0 + X^1 + X^2 + X^3 + \ldots\ldots$$

$$X^+ = X^1 + X^2 + X^3 + \ldots\ldots$$

- $X\ X^* = X^*\ X = X^+$
- $a\ a^* = a^*\ a = a^+$

$$X^{\underline{n}} = X^0 + X^1 + X^2 + X^3 + \ldots\ldots + X^n$$

Path Products & expressions – Path products

More Rules…

$$X^{\underline{m}} + X^{\underline{n}} = X^{\underline{n}} \quad \text{if } n \geq m$$
$$= X^{\underline{m}} \quad \text{if } n < m$$
: **Rule 6**

$$X^{\underline{m}} X^{\underline{n}} = X^{\underline{m+n}}$$
: **Rule 7**

$$X^{\underline{n}} X^{*} = X^{*} X^{\underline{n}} = X^{*}$$
: **Rule 8**

$$X^{\underline{n}} X^{+} = X^{+} X^{\underline{n}} = X^{+}$$
: **Rule 9**

$$X^{*} X^{+} = X^{+} X^{*} = X^{+}$$
: **Rule 10**

Identity Elements ..

1 : Path of Zero Length

$1 + 1 = 1$: **Rule 11**

$1X = X1 = X$: **Rule 12**

$1^{\underline{n}} = 1^{n} = 1^{*} = 1^{+} = 1$: **Rule 13**

$1^{+} + 1 = 1^{*} = 1$: **Rule 14**

Identity Elements ..

0 : empty set of paths

$X + 0 = 0 + X = X$: **Rule 15**

$X\,0 = 0\,X = 0$: **Rule 16**

$0^* = 1 + 0 + 0^2 + 0^3 + \ldots = 1$: **Rule 17**

Path Products & expressions – Reduction Procedure

- To convert a flow graph into a path expression that denotes the set of all entry/exit paths.

- Node by Node Reduction Procedure

Path Products & expressions – Reduction Procedure

Initialization Steps:

1. Combine all serial links by multiplying their path expressions.

1. Combine all parallel links by adding their path expressions.

1. Remove all self-loops - replace with links of the form X*

Steps in the Algorithm's loop

4. Select a non-initial & non-final node.

Replace it with a set of equivalent links, whose path expressions correspond to all the ways you can form a product of the set of in-links with the set of out-links of that node.

Path Products & expressions – Reduction Procedure

Steps in the Algorithm's loop:

5. Combine any serial links by multiplying their path expressions.
   (as in step 1)

6. Combine any parallel links by adding their path expressions.
   (as in step 2)

7. Remove all the self-loops.
   (as in step 3)

8. IF there's just one node between entry & exit nodes, path expression for the flow graph is the link's path expression. ELSE, return to step 4.

Path Expression for a Flow Graph

- is not unique
- depends on the order of node removal.

Path Products & expressions – Reduction Procedure Example

Cross-Term Step (Step 4 of the algorithm)

- Fundamental step.
- Removes nodes one by one till there's one entry & one exit node.
- Replace the node by path products of all in-links with all out-links and interconnecting its immediate neighbors.

Path Products & expressions – Reduction Procedure Example

Processing of Loop Terms:

Path Products & expressions – Reduction Procedure Example

Processing of Loop Terms:

Path Products & expressions – Reduction Procedure Example

Processing of Loop Terms:

a (bd)* bc (e (bd)* bc)* f

Path Products & expressions – Reduction Procedure Example

Example:

Path Products & expressions – Reduction Procedure Example

Path Products & expressions – Reduction Procedure Example

Path Products & expressions – Reduction Procedure Example

(ilhd)*imf(bgif)*b(c+gkh)d

Path Products & expressions – Reduction Procedure Example

Flow Graph Path Expression :

a(bgif)*b(c+gkh)d {(ilhd)*imf(bgif)*b(c+gkh)d}* (ilhd)*e

Path Products & expressions – Before going into Applications

Before that, we learn:

Identities

Structured Flow Graphs (code/routines)

Unstructured Flow graphs (routines)

Path Products & expressions – Identities / Rules

$(A + B)^*$	$=$	$(A^* + B^*)^*$	: I1
	$=$	$(A^* \; B^*)^*$	: I2
	$=$	$(A^* B)^* \; A^*$	: I3
	$=$	$(B^* A)^* \; B^*$	: I4
	$=$	$(A^* B + A)^*$	: I5
	$=$	$(B^* A + B)^*$	: I6
$(A + B + C + \ldots)^*$	$=$	$(A^* + B^* + C^* + \ldots)^*$	: I7
	$=$	$(A^* B^* C^* \ldots)^*$	: I8

Derived by removing nodes in different orders & applying the series-parallel-loop rules.

Path Products & expressions – Structured Flow Graphs

Reducible to a single link by successive application of the transformations shown below.

Structured flow graph transformations

Properties:

- No cross-term transformation.
- No GOTOs.
- No entry into or exit from the middle of a loop.

Path Products & expressions – Structured Flow Graphs

Some examples:

Path Products & expressions – UNstructured Flow Graphs

Some examples – unstructured flow graphs/code:

Jumping into loops

Jumping out of loops

Branching into Decisions

Branching out of Decisions

Logic based testing: Overview

- Logic is used in a program by programmers. Boolean algebra is the way to work with logic – simplification & calculation.
- **Hardware logic testing** – hardware logic test design tools and methods use logic & Boolean algebra. Hardware design language compilers/translators use logic & Boolean algebra.
- Impact of errors in specifications of a software is high as these are first in and last out. So, higher level language for specs is desired to reduce the number of errors. Higher order logic systems are used for formal specifications. The tools to simplify, transform and check specs use Boolean algebra.

Logic based testing: Overview

Knowledge based systems:

- Knowledge based systems and artificial intelligence systems use high level logic languages which are based on rule bases consisting of rules. Rules are predicate expressions containing domain knowledge related elements combined with logical connectives. The answers to queries (problems) are derived based on Boolean algebraic operations performed on the rule bases. Such programs are called inference engines.

Modeling Logic with Decision Tables

- A matrix representation of the logic of a decision
- Specifies the possible conditions and the resulting actions
- Best used for complicated decision logic

Modeling Logic with Decision Tables

- Consists of three parts
  - Condition stubs
    - Lists condition relevant to decision
  - Action stubs
    - Actions that result from a given set of conditions
  - Rules
    - Specify which actions are to be followed for a given set of conditions

Modeling Logic with Decision Tables

- Indifferent Condition
  - Condition whose value does not affect which action is taken for two or more rules
- Standard procedure for creating decision tables
  - Name the condition and values each condition can assume
  - Name all possible actions that can occur
  - List all rules
  - Define the actions for each rule
  - Simplify the table

Complete decision table for payroll system example

	Conditions/ Courses of Action	Rules					
		1	2	3	4	5	6
Condition Stubs	Employee type	S	H	S	H	S	H
	Hours worked	<40	<40	40	40	>40	>40
Action Stubs	Pay base salary	X		X		X	
	Calculate hourly wage		X		X		X
	Calculate overtime						X
	Produce Absence Report		X				

Constructing a Decision Table

- PART 1. FRAME THE PROBLEM.
  - Identify the conditions (decision criteria). These are the factors that will influence the decision.
    - E.g., We want to know the total cost of a student's tuition. What factors are important?
  - Identify the range of values for each condition or criteria.
    - E.g. What are they for each factor identified above?
  - Identify all possible actions that can occur.
    - E.g. What types of calculations would be necessary?

Constructing a Decision Table

- PART 2. CREATE THE TABLE.
  - Create a table with 4 quadrants.
    - Put the conditions in the upper left quadrant. One row per condition.
    - Put the actions in the lower left quadrant. One row per action.
  - List all possible rules.
    - Alternate values for first condition. Repeat for all values of second condition. Keep repeating this process for all conditions.
    - Put the rules in the upper right quadrant.
  - Enter actions for each rule
    - In the lower right quadrant, determine what, if any, appropriate actions should be taken for each rule.
  - Reduce table as necessary.

Example

- Calculate the total cost of your tuition this quarter.
  - What do you need to know?
    - Level. (Undergrad or graduate)
    - School. (CTI, Law, etc.)
    - Status. (Full or part time)
    - Number of hours
  - Actions?

- Actions?
  - Consider CTI only (to make the problem smaller):
    - U/G
      - Part Time (1 to 11 hrs.): $335.00/per hour
      - Full Time (12 to 18 hrs.): $17,820.00
      - * Credit hours over 18 are charged at the part-time rate
    - Graduate:
      - Part time (1 to 7 hrs.): $520.00/per hour
      - Full time (>= 8 hrs.): $520.00/per hour
- Create a decision table for this problem. In my solution I was able to reduce the number of rules from 16 to 4.

Boolean Algebra

A *Boolean algebra* consists of:

- a set $B=\{0, 1\}$,
- 2 binary operations on B (denoted by $+$ & $\times$),
- a unary operation on B (denoted by $'$), such that :

$0 + 0 = 0$	$0 \times 0 = 0$
$1 + 0 = 1$	$0 \times 1 = 0$
$0 + 1 = 1$	$1 \times 0 = 0$
$1 + 1 = 1$	$1 \times 1 = 1$

$0' = 1$ and $1' = 0$.

Rules of a Boolean Algebra

The following axioms ('rules') are satisfied for all elements x, y& z of B:

(1) $x + y = y + x$ (commutative axioms)

$x \times y = y \times x$

(2) $x + (y + z) = (x + y) + z$ (associative axioms)

$x \times (y \times z) = (x \times y) \times z$

(3) $x \times (y + z) = (x \times y) + (x \times z)$

$x + (y \times z) = (x + y) \times (x + z)$ (distributive axioms)

(4) $x + 0 = x$ $x \times 1 = x$ (identity axioms)

(5) $x + x' = 1$ $x \times x' = 0$ (inverse axioms)

Laws of Boolean Algebra

- In addition to the laws given by the axioms of Boolean Algebra, we can show the following laws

$x'' = x$		(double complement)
$x + x = x$	$x \times x = x$	(idempotent)
$(x + y)' = x' \times y'$	$(x \times y)' = x' + y'$	(de Morgan's laws)
$x + 1 = 1$	$x \times 0 = 0$	(annihilation)
$x + (x \times y) = x$	$x \times (x + y) = x$	(absorption)
$0' = 1$	$1' = 0$	(complement)

Exercise

Simplify the Boolean expression

$$(x' \times y) + (x \times y)$$

Solution: $(x' \times y) + (x \times y)$

$= (y \times x') + (y \times x)$ (commutative)

$= y \times (x' + x)$ (distributive)

$= y \times (x + x')$ (commutative)

$= y \times 1$ (inverse)

$= y$ (identity)

Thus $(x' \times y) + (x \times y) = y$

Boolean Notation

- This means that in effect we'll be employing Boolean Algebra notation.
- The truth tables can be rewritten as

x	y	$x+y$	$x \times y$	x'
0	0	0	0	1
0	1	1	0	
1	0	1	0	0
1	1	1	1	

Notational Short-cuts

We will employ short-cuts in notation:

(1) In 'multiplication' we'll omit the symbol $\times$, & write xy for $x \times y$ (just as in ordinary algebra)

(2) The associative law says that

$$x + (y + z) = (x + y) + z$$

So we'll write this as simply $x + y + z$, because the brackets aren't necessary.

Notational Short-cuts

Similarly, write the product of 3 terms as xyz

(3) In ordinary algebra, the expression $x + y \times z$ means $x + (y \times z)$, because of the convention that *multiplication takes precedence over addition.*

e.g. $x + yz$ means $x + (y \times z)$, and *not* $(x + y) \times z$
Similarly, $ab + cd$ means $(a \times b) + (c \times d)$

Reducing Boolean Expressions

- Is this the smallest possible implementation of this expression? No!

$$G = xyz + xyz' + x'yz$$

- Use Boolean Algebra rules to reduce complexity while preserving functionality.

- Step 1: Use idempotent law (a + a = a). So

$$xyz + xyz' + x'yz = xyz + xyz + xyz' + x'yz$$

Reducing Boolean Expressions

- Step 2: Use distributive law $a(b + c) = ab + ac$. So $xyz + xyz + xyz' + x'yz = xy(z + z') + yz(x + x')$

- Step 3: Use Inverse law $(a + a' = 1)$. So

  $$xy(z + z') + yz(x + x') = xy.1 + yz.1$$

- Step 4: Use Identity law $(a . 1 = a)$. So

  $$xy + yz = xy.1 + yz.1 = xyz + xyz' + x'yz$$

Karnaugh maps

- Alternate way of representing Boolean function
  - All rows of truth table represented with a square
  - Each square represents a minterm

x \ y	0	1
0	x'y'	x'y
1	xy'	xy

x \ y	0	1
0	1	1
1	0	0

x	y	F
0	0	1
0	1	1
1	0	0
1	1	0

Karnaugh maps

- Easy to convert between truth table, K-map, and SOP.
  - Unoptimized form: number of 1's in K-map equals number of minterms (products) in SOP.
  - Optimized form: reduced number of minterms

$$F(x,y) = x'y + x'y' = x'$$

Karnaugh Maps

- A Karnaugh map is a graphical tool for assisting in the general simplification procedure.
- Two variable maps.

A \ B	0	1
0	0	1
1	1	0

$F = AB' + A'B$

A \ B	0	1
0	0	1
1	1	1

$F = AB + A'B + AB'$

- Three variable maps.

A \ BC	00	01	11	10
0	0	1	0	1
1	1	1	1	1

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$F = AB'C' + AB'C + ABC + ABC' + A'B'C + A'BC'$

Rules for K-Maps

- We can reduce functions by circling 1's in the K-map.
- Each circle represents minterm reduction.
- Following circling, we can deduce minimized and-or form.

x \ y	0	1
0	1	1
1	0	0

$F(x,y) = x'y' + x'y' = x'$

Rules for K-Maps

Rules to consider

1. Every cell containing a 1 must be included at least once.

2. The largest possible “power of 2 rectangle” must be enclosed.

Karnaugh Maps

- A Karnaugh map is a graphical tool for assisting in the general simplification procedure.
- Two variable maps.

A \ B	0	1
0	0	1
1	1	0

$F = AB' + A'B$

A \ B	0	1
0	0	1
1	1	1

$F = AB + A'B + AB'$

$F = A + B$

- Three variable maps.

A \ BC	00	01	11	10
0	0	1	0	1
1	1	1	1	1

$F = A + B'C + BC'$

$F = AB'C' + AB'C + ABC + ABC' + A'B'C + A'BC'$

More Karnaugh Map Examples

Examples

b \ a	0	1
0	0	1
1	0	1

f = a

b \ a	0	1
0	1	1
1	0	0

g = b'

c \ ab	00	01	11	10
0	0	0	1	0
1	0	1	1	1

cout = ab + bc + ac

c \ ab	00	01	11	10
0	0	0	1	1
1	0	0	1	1

f = a

1. Circle the largest groups possible.
2. Group dimensions must be a power of 2.
3. Remember what circling means

Specifications

Specification validation procedure / steps

1. Rewrite the specifications using consistent terminology.

2. Identify the predicates on which the cases are based. Name them with suitable letters, such as A, B, C.

3. Rewrite the specification in English that uses only the logical connectives AND, OR, and NOT, however stilted it may seem.

4. Convert the rewritten specifications into an equivalent set of Boolean expressions.

5. Identify the default action and cases, if any are specified.

Specifications Continue

6. Enter the Boolean expressions in a KV chart and check for consistency. If the specifications are consistent, there will be no overlaps, except for the cases that result in multiple actions.
7. Enter the default cases, and check for consistency.
8. If all boxes are covered, the specification is complete.
9. If the specification is incomplete or inconsistent, translate the corresponding boxes of the KV chart back into English and get a clarification, explanation, or revision.
10. If the default cases were not specified explicitly, translate the default cases back into English and get a confirmation.

UNIT-V

State, State Graphs and Transition testing: State graphs, good & bad state graphs, state testing, Testability tips.

Graph Matrices and Application: Motivational overview, matrix of graph, relations, power of a matrix, node reduction algorithm, building tools.

OVER VIEW

- A state graph and its associated state table are useful models for describing software (program) behavior. The finite state machine can be used as a functional (behavioral) testing tool as well as a tool for designing a testable program.

state graph

- A **state graph** is a graphical representation of the program **(its FSM)** in terms of states, transitions, inputs and outputs (erroneous or normal). It has one start state and usually, an end/destination/exit state.
- Note => In the exam you may draw only 3 state graph for simplicity.
- State graph in the above example is used to model the behavior of the program that recognizes a string occurrence at the input. It can be used to design, implement and the testing of the program

A Property of a state graph

- State graphs are not dependent on time or temporal behavior or the program. (Temporal behavior is represented by some time sequence diagrams etc..) The system changes state only when an event (with an input sequence occurs or an epsilon symbol representing no event appears at the input of a transition).
- State graphs (FSM) are implemented as state tables which are represented in software with definite data structures and associated operations.

State table

Very big state graphs are difficult to follow as the diagrams get complicated and links get entwined. It is more convenient to represent the state graph as a table called state table or state transition table.

Each row represents the transitions from the originating state. There is one column for each input symbol (erroneous input or normal input). The entry in the table represents the new state to which the system transits to on this transition and the output it prints on the target printer device or on the output side.

A Property of a state graph

- State graphs are not dependent on time or temporal behavior or the program. (Temporal behavior is represented by some time sequence diagrams etc..) The system changes state only when an event (with an input sequence occurs or an epsilon symbol representing no event appears at the input of a transition).
- State graphs (FSM) are implemented as state tables which are represented in software with definite data structures and associated operations.

Software implementation of state table

- There are four tables that are needed.

1. A table or a process that encodes the input values into a compact list (INPUT_CODE_TABLE)
2. A table that specifies the next state for every combination of state and input code. (TRANISITION_TABLE)
3. A table or case statement that specifies the output (or output code) associated with every state-input combination (OUTPUT_TABLE)
4. A table that stores the present state of each device or process or component or system that uses the same state table. (DEVICE_TABLE)

Software implementation of state table

The process of usage of state table is as follows:

1. the present state is fetched from the memory (from DEVICE_TABLE).
2. the present input value (symbol) is fetched from the environment. It is encoded if it is non-numerical by using the INPUT_CODE_TABLE.
3. The present state and the input code are concatenated to give a pointer (row,column) into a cell of the TRANSITION_TABLE.
4. The OUTPUT_TABLE contains a pointer to the routine to be executed when that state-input combination occurs.
5. The same pointer value is used to fetch the new state value, which is then stored in DEVICE_TABLE

Principles of state testing

As it is not possible to test every path thru a state graph, use the notion of coverage. We assume that the graph is strongly connected.

1. It is not useful or practical to plan an entire grand tour of the states for testing initially as it does not work out due to possibility of bugs.
2. During the maintenance phase only few transitions and states need to be tested which are affected.

3 For very long test input symbol sequences it is difficult to test the system.

Uses/Advantages of state testing

- State testing can find bugs which are not possible to be found with other types of testing. Normally most of systems can be modeled as state graphs.
- It can find if the specifications are complete and ambiguous. This is seen clearly if the state table is filled with multiple entries in some cells or some cells are empty. IT can also tell if some default transitions or transitions on erroneous inputs are missing.
- State testing can identify the system's seemingly impossible states and checks if there are transitions from these states to other states are defined in the specifications or not. That is, the error recovery processes are defined for such impossible states.
-

Uses/Advantages of state testing

- State testing can simplify the design of the program / system by identifying some equivalent states and then merging these states. Also, state testing using FSM can allow design/test design in a hierarchical manner if the state tables are so designed.
- The state testing can identify if the system reaches a dead state / unreachable states and allow one to correct the program specifications and make the system complete, robust and consistent.
- The bugs in the functional behavior can be caught earlier and will be less expensive if state testing is done earlier than the structural (white box) testing.

DisAdvantages of state testing

- Temporal behavior is not tested.
- There could be encoding errors in inputs, outputs, states, input-state combinations, identifying the number of states and merger of equivalent states. All these errors are not always easy to detect and correct.
- State transition testing does not guarantee the complete testing of the program. How much of testing with how many combinations of input symbol sequences constitutes sufficient number of tests is not clear/known. It is not practical to test thru every path in the state graph.
- Functional behavior is tested and structural bugs are not tested for. There could be difficulty if those bugs are found and mixed up with behavioral bugs.
- We assume that the state graph is strongly connected that is every node is connected to every other node thru a path.

DisAdvantages of state testing

- Temporal behavior is not tested.
- There could be encoding errors in inputs, outputs, states, input-state combinations, identifying the number of states and merger of equivalent states. All these errors are not always easy to detect and correct.
- State transition testing does not guarantee the complete testing of the program. How much of testing with how many combinations of input symbol sequences constitutes sufficient number of tests is not clear/known. It is not practical to test thru every path in the state graph.
- Functional behavior is tested and structural bugs are not tested for. There could be difficulty if those bugs are found and mixed up with behavioral bugs.
- We assume that the state graph is strongly connected that is every node is connected to every other node thru a path.

DisAdvantages of state testing

- Temporal behavior is not tested.
- There could be encoding errors in inputs, outputs, states, input-state combinations, identifying the number of states and merger of equivalent states. All these errors are not always easy to detect and correct.
- State transition testing does not guarantee the complete testing of the program. How much of testing with how many combinations of input symbol sequences constitutes sufficient number of tests is not clear/known. It is not practical to test thru every path in the state graph.
- Functional behavior is tested and structural bugs are not tested for. There could be difficulty if those bugs are found and mixed up with behavioral bugs.
- We assume that the state graph is strongly connected that is every node is connected to every other node thru a path.

Application areas for state testing using FSM model

Any program that processes input as a sequence of events/symbols and produces output such as detection of specified input symbol combinations, sequential format verification, parsing, etc. (compilers & translators).

- Communication Protocols: processing depends on current state of the
- protocol stack, OS, network environment and the message received
- concurrent systems,
- system failures and the corresponding recovery systems,
- distributed data bases,
- device drivers – processing depends on the state of the device &

Application areas for state testing using FSM model

- operation requested by the user or system
  - multi-tasking systems,
  - human computer interactive systems,
  - resource management systems – processing depends on availability
- levels and states of resources
  - Processing of hierarchical pop-up menus on windows based software systems – letting the user navigate thru menus
  - the web based application software, embedded systems and other systems also use this model for design and testing.

Good-state graphs and Bad state graph

The principles of judging whether a state graph is good or bad are:

- the total number of states is equal to the product of possibilities of factors that make up the state. (ie., number of permutations of all values of all attributes/properties of the system/component)
- For every state and every input there is exactly one transition specified to exactly one, possibly the same, state.
- For every transition there is one output action specified. That output could be trivial (epsilon), but at least one output does some thing sensible.
- For every state there is a sequence of inputs that drives the system to the starting (same) state.
- A good state graph has at least two input symbols. With one symbol only a limited number of useful graphs are possible.
- Bad state graphs contain states not reachable. It is not possible to reach every state from every other state. It is not possible to reach start state from itself.

Good-state graphs and Bad state graph

- A good state graph has at least two input symbols. With one symbol only a limited number of useful graphs are possible.
- Bad state graphs contain states not reachable. It is not possible to reach every state from every other state. It is not possible to reach start state from itself.

OVER VIEW

- The graphs (flow/directed) are basically behavioral representation of software program. Many problems/applications we want to have solution for :

- – to find set of covering paths, set of values that will sensitize the paths, logic function that controls the flow, the processing time of the routine, the equations that define a domain, whether the routine pushes or pops, or whether the state is reachable or not etc.. -

- usually necessitate path tracing thru the graph. Most of the testing and design strategies use graphs to represent the program and need some coverage to be obtained.

- Path tracing thru pictorial graphs and by hand is error prone, difficult to handle and often confusing when the program is large.

- Graph representation by a matrix enables matrix operations equivalent to path tracing to be used. These matrix operations are methodical and mechanical and hence are more reliable. Also, these can be implemented by software tools & automated enabling a good solution for the questions/issues that interest a designer and a tester.

Figure **A**

MATRIX REPRESENTATION:OVER VIEW

- A graph matrix is a square array with one row and one column for every node in the graph. Each row-column combination represents a relation between the node corresponding to the row i and the node corresponding to the column j.

- The relation could be just the link name if there is a direct link between the two nodes. The matrix is a square matrix of size n (where n is the # of nodes). Self loops are shown as diagonal entries.

- Actually, an element **A [i , j]** is a path expr between node i to j (sum of parallel path products between the nodes.)

RELATION MATRIX

A matrix representation defined as just above.

Corresponding relation Matrix is

$$
\begin{array}{cccccccc}
\textbf{node} & & 1 & 2 & 3 & 4 & 5 & \\
\textbf{1} & [& 0 & 0 & a & 0 & 0 &] \\
\textbf{2} & [& 0 & 0 & 0 & 0 & 0 &] \\
\textbf{3} & [& 0 & d & 0 & b & 0 &] \\
\textbf{4} & [& 0 & c & 0 & 0 & f &] \\
\textbf{5} & [& 0 & g & e & 0 & h &]
\end{array}
$$

CONNECTION MATRIX

The relation in a connection matrix is defined as follows:

A [i, j] = 1 if there is a direct link from i to j.
0 otherwise

It is the relation matrix with a non zero entry replaced by **1**.

Corresponding connection Matrix is

$$\begin{array}{c} \text{node} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \end{bmatrix} \end{array}$$

Maths / Algebra over the relation **connection**:

1 X 1 = 1 0 X 0 = 0 1 X 0 = 0 0 X 1 = 0

A [i,j] x A [j,k] represents existence of a (directed) path from i to k

Each entry in the row **r** represents an outgoing link from the node **r**. Each entry in the column **c** represents an incoming link into the node **c**. A branch node **r** has >1 non zero entries in its row **r**. A junction node c has > 1 non-zero entries in its column.

CYCLOMATIC COMPLEXITY

Is 1 plus the number of binary decisions that are present in a given program / graph. The use of the cyclomatic complexity is in estimation of efforts needed for design & testing of a given software program.

calculated as follows

Prepare the relation matrix and from that prepare the connection matrix. For each row (represents branches ie., decisions) obtain the sum of the entries (each entry is 1 or 0). Number of binary decisions at this node / row r is equal to the sum - 1 as number of binary decisions at a node = number of outgoing branches - 1.

Sum the number of binary decisions of each row and add 1 to get the cycl. Complexity.

EXAMPLE:

First row : total = 1 then minus 1 => 0

2nd row	0	nil	0
3rd row	2	1	1
4th row	2	1	1
5th row	3	1	2

		Total	4

example:

cyclomatic complexity = 4+1 = 5

$$A[i, j] = \sum_{K=1 \text{ to } n} a_{ik}\, a_{kk}\, a_{kj}$$

RELATIONSHIPS

A relation is a property that exists between two elements / objects of interest. For example, a R b could denote the relation R as existence of a directed link from a to b.

Redefinitions of graphs:

A graph consists of a set of abstract objects called nodes and a (binary) relation R between (among pairs of) the nodes. If a R b, it means that node a has the relation to node b – which denoted by a link from a to b.

PROPERTIES OF RELATIONSHIPS

1. A relation R is **transitive i**f a R b and b R c imply a R c. Examples of transitive relations are **is connected to** , >= etc.
2. Examples for **an Intransitive relation** are is acquainted with, is a neighbor of, not equal to etc..
3. A relation R is reflexive if for every a, a R a. It is equivalent to a self loop at every node a in the graph.
4. Examples of **irreflexive relation** are not equals, is under etc.
5. A relation is **a Symmetric relation** if for every a and b , a R b implies b R a. It means that if there is a link from a to b then there is a link between b to a. Then it is as good as an undirected graph (unless the link weights are different for the different directions).
6. Examples of symmetric relation are : is a relative of, is a friend of, equals , not equals etc..
7. A relation R is **Anti-symmetric** relation if for every a and b, if **a R b** and **b R a** imply that a = b. That is there is no symmetry between any two different elements.
8. **An equivalence relation** is a relation that satisfies the reflexive, transitive and symmetric properties. Example : is equal to.

POWERS OF GRAPH MATRIX

A (connection or relation) graph matrix product represents existence of a path or its path expression respectively between two nodes with a common intermediate node.

$$C = A \times B \qquad C[i, j] \text{ or } c_{ii} = \sum_{K=1 \text{ to } n} \mathbf{a}_{ik} \, \mathbf{b}_{kj}$$

- $\sum \mathbf{a}_{ik} \, \mathbf{b}_{kj}$ implies set (sum) of all paths between I and j thru all intermediate nodes k in the graph.

A2 represents the set of all path segments of length two (links) ie., each non-zero element A [i , j] is the path expression for the set of 2 link long path segments from node i to j.

An-1 represents the set of all path segments of length n-1 (links) ie., each non-zero element A [i , j] is the path expression for the set of n-1 link long path segments from node i to j.

Graph matrix product is not necessarily commutative, but is associative and distributive. The non-zero elements along principal diagonal represent self-loops on those nodes

PARTITIONING ALGORITHM – GROUPING LOOPS AND OBTAINING LOOP-FREE GRAPH

We will consider a graph (over a transitive relation). The graph may have loops. We would like to partition the graph by grouping the nodes in such a way that every node is contained with in one group of another. Such a graph is partially ordered.

The main use is to embed the loops into subroutines to have a resulting loop-free flow in the program/graph. This enables a better design and analysis and testing design.

Here we use the **connection matrix**. To do grouping of loops we just need to calculate intersection of the transitive closure of graph matrix with itself.

Ie., find (A + **I**) n # (A + **I**) nT **Remember this**

If n = 5 then calculate A and its square, then its square and multiply with A. Then obtain its transpose and then do intersection. Here, the intersection means binary AND directly element a [i, j I by element a [i,j].

Once we calculate the intersection as above, the rows or columns which are identical will form a group. Replace them by one row/column. Now redraw the graph with reduced number of nodes. Each group is a merger of the loop-forming node set. The result will be a loop-free directed graph which is partially ordered.

Example: PARTITIONING ALGORITHM

Take an example with just 3 nodes only.

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad A+I = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \quad (A+I)2 = \begin{matrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{matrix} \quad (A+I)3 = \begin{matrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{matrix}$$

Now find (A+I)3 Transpose $= \begin{matrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$ Intersection result $= \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{matrix}$

As you see columns 2,3 and rows 2,3 are identical. Merge the two nodes.
You have now the following with node 2,3 engulfing & enclosing the loop.

NODE REDUCTION ALGORITHM

- The advantage of this node reduction algorithm using graph matrices is that we do not need to redraw the graph after reduction of a node. Calculations are done methodically and can be automated. It is actually matrix equivalent of the node reduction algorithm in the chapter 'paths, path expressions & regular expr..'.

- **Steps are :**

1. select a node for removal Other than start or end node. Replace the node by equivalent links that bypass that node and add those links to the links they parallel.
2. Combine the parallel terms and simplify as much as you can.
3. Observe loop terms and adjust the outlinks of every node that had a self-loop to account for the effect of the loop.
4. the result is a matrix whose size has been reduced by 1. Continue until the two nodes **entry** and **exit** exist.

THE ALGORITHM:

Step 1 : Before step1, Always first eliminate the self nodes ie., removal of a non-zero diagonal entry **d**. Multiply each path expr. in the row r with d* where d is a non-zero diagonal element in row/column r. That is, multiply each outgoing link from node r with d* . Now replace d with 0.

Step 2: Select the last column **c**. Take one non-zero element A [r, c] in column c. Pre-Multiply each entry A [c, j] in the (last) row **c** with A[r, c]. Add the result to the element in the intersection of c and j , ie., to A [r, j]. Repeat this for every non-zero element in the last column. Now, remove the last row & column.

Step3: actually amounts to multiplying each incoming link into a node with each outgoing link from that node and then replacing those two links by a direct path bypassing this last node. In the above A [r, c] is an incoming link into node c and A [c, j] is an outgoing link from node c. Multiply and add the result to path expr. from node r to node j ie., A [r, j].

Application Of Node Reduction Algorithm

a. maximum number of paths – to find

use the relation matrix with the entry being the number of paths between a & b. When you multiply and add, use the arithmetic rules defined for this property.

b. Probability of getting there

Use the probability rules for the product and sum. Use the relation matrix with the entries being the weights (probability of getting to the neighboring node from a node).

c. Get/ Return or Push/Pop problem

Use the same arithmetic rules for G & R as defined earlier in path expressions, regular expressions chapter.

Building software tools using graph matrices for implementation of node reduction algorithm / partitioning algorithm

1. **representation of graph matrix – array / linked list**
2. **issues, merits & demerits of the above representations.**

The degree of a node is the sum of out-degree and in-degree. Usually the degree of a node in a good program is around 3 to 4. We use arrays or linked lists. There are advantages of representation by linked lists over arrays. These are :

1. space grows as O(n2). For a linked list it is only k * n where k is avg degree of a node which is 3 to 4.
2. Weights are complicated for the links and have several components. So would require an additional array for link weights.
3. Variable length weights – as the weights can be expressions or numbers or others, we need a 2-d string array representation for link weight array. Most of the entries are null as graph matrix is sparse usually.
4. Processing time – As each array element is processed whether null or not, the processing time for the null elements could be significant as there are many of them usually in a graph matrix.

EXAMPLE

EXAMPLE-CONTINUE…

<u>Linked list representation</u>

linked List entry (one node in the listed list)	**its content**
node 1,3; a	
2	node 2, exit
3	node 3, 2 ; d
	3, 4 ; b
4	node 4, 2 ; c
	4, 5 ; f
5	node 5, 2 ; g
	5 ,3 ; e
	5 ,5 ; h

Corresponding relation Matrix is

$$
\begin{array}{cc}
\text{node} & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\
\begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} &
\begin{bmatrix}
0 & 0 & a & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & d & 0 & b & 0 \\
0 & c & 0 & 0 & f \\
0 & g & e & 0 & h
\end{bmatrix}
\end{array}
$$

EXAMPLE-CONTINUE…

<u>Linked list representation</u>

linked List entry (one node in the listed list)	**its content**
node 1,3; a	
2	node 2, exit
3	node 3, 2 ; d
	3, 4 ; b
4	node 4, 2 ; c
	4, 5 ; f
5	node 5, 2 ; g
	5 ,3 ; e
	5 ,5 ; h

Corresponding relation Matrix is

$$
\begin{array}{cc}
\text{node} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} &
\begin{bmatrix}
0 & 0 & a & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & d & 0 & b & 0 \\
0 & c & 0 & 0 & f \\
0 & g & e & 0 & h
\end{bmatrix}
\end{array}
$$