

FOUNDATIONS OF
SOFTWARE
TESTING | 2E



ADITYA P. MATHUR

FOUNDATIONS OF SOFTWARE TESTING

SECOND EDITION

Fundamental Algorithms and Techniques

An Undergraduate and Graduate Text
A Reference for the Practicing Software Engineer

ADITYA P. MATHUR
Purdue University

PEARSON

Delhi • Chennai

*In Memory of
My Loving Mother and Father
My Friend (Late) Dr. Ranjit Singh Chauhan*

Contents

Preface to the Second Edition

Preface to the First Edition

Acknowledgements

PART I: PRELIMINARIES

1. Preliminaries: Software Testing

1.1. Humans, Errors, and Testing

1.1.1 Errors, faults, and failures

1.1.2 Test automation

1.1.3 Developer and tester as two roles

1.2. Software Quality

1.2.1 Quality attributes

1.2.2 Reliability

1.3. Requirements, Behavior, and Correctness

1.3.1 Input domain

1.3.2 Specifying program behavior

1.3.3 Valid and invalid inputs

1.4. Correctness Versus Reliability

1.4.1 Correctness

1.4.2 Reliability

1.4.3 Operational profiles

1.5. Testing and Debugging

1.5.1 Preparing a test plan

1.5.2 Constructing test data

1.5.3 Executing the program

1.5.4 Assessing program correctness

1.5.5 Constructing an oracle

1.6. Test Metrics

1.6.1 Organizational metrics

1.6.2 Project metrics

1.6.3 Process metrics

1.6.4 Product metrics: generic

1.6.5 Product metrics: OO software

1.6.6 Progress monitoring and trends

1.6.7 Static and dynamic metrics

1.6.8 Testability

1.7. Software and Hardware Testing

1.8. Testing and Verification

1.9. Defect Management

1.10. Test Generation Strategies

1.11. Static Testing

1.11.1 Walkthroughs

1.11.2 Inspections

1.11.3 Software complexity and static testing

1.12. Model-Based Testing and Model Checking

1.13. Types of Testing

1.13.1 Classifier: C1: Source of test generation

1.13.2 Classifier: C2: Life cycle phase

1.13.3 Classifier: C3: Goal-directed testing

1.13.4 Classifier: C4: Artifact under test

1.13.5 Classifier: C5: Test process models

1.14. The Saturation Effect

1.14.1 Confidence and true reliability

1.14.2 Saturation region

1.14.3 False sense of confidence

1.14.4 Reducing Δ

1.14.5 Impact on test process

1.15. Principles of Testing

1.16. Tools

Summary

Exercises

2. Preliminaries: Mathematical

2.1. Predicates and Boolean Expressions

2.2. Control Flow Graph

2.2.1 Basic blocks

2.2.2 Flow graphs

2.2.3 Paths

2.2.4 Basis paths

2.2.5 Path conditions and domains

2.2.6 Domain and computation errors

2.2.7 Static code analysis tools and static testing

2.3. Execution History

2.4. Dominators and Post-Dominators

2.5. Program Dependence Graph

2.5.1 Data dependence

2.5.2 Control dependence

2.5.3 Call graph

2.6. Strings, Languages, and Regular Expressions

2.7. Tools

Summary

Exercises

PART II: TEST GENERATION

3. Domain Partitioning

3.1. Introduction

3.2. The Test Selection Problem

3.3. Equivalence Partitioning

3.3.1 Faults targeted

3.3.2 Relations

3.3.3 Equivalence classes for variables

3.3.4 Unidimensional partitioning versus multidimensional partitioning

3.3.5 A systematic procedure

3.3.6 Test selection

3.3.7 Impact of GUI design

3.4. Boundary Value Analysis

3.5. Category-partition Method

3.5.1 Steps in the category-partition method

Summary

Exercises

4. Predicate Analysis

4.1. Introduction

4.2. Domain Testing

4.2.1 Domain errors

4.2.2 Border shifts

4.2.3 ON-OFF points

4.2.4 Undetected errors

4.2.5 Coincidental correctness

4.2.6 Paths to be tested

4.3. Cause-effect Graphing

4.3.1 Notation used in cause-effect graphing

4.3.2 Creating cause-effect graphs

4.3.3 Decision table from cause-effect graph

4.3.4 Heuristics to avoid combinatorial explosion

4.3.5 Test generation from a decision table

4.4. Tests Using Predicate Syntax

4.4.1 A fault model

4.4.2 Missing or extra Boolean variable faults

4.4.3 Predicate constraints

4.4.4 Predicate testing criteria

4.4.5 BOR, BRO, and BRE adequate tests

4.4.6 BOR constraints for non-singular expressions

4.4.7 Cause-effect graphs and predicate testing

4.4.8 Fault propagation

4.4.9 Predicate testing in practice

4.5. Tests Using Basis Paths

4.6. Scenarios and Tests

Summary

Exercises

5. Test Generation From Finite State Models

5.1. Software Design and Testing

5.2. Finite State Machines

5.2.1 *Excitation using an input sequence*

5.2.2 *Tabular representation*

5.2.3 *Properties of FSM*

5.3. Conformance Testing

5.3.1 *Reset inputs*

5.3.2 *The testing problem*

5.4. A Fault Model

5.4.1 *Mutants of FSMs*

5.4.2 *Fault coverage*

5.5. Characterization Set

5.5.1 *Construction of the k-equivalence partitions*

5.5.2 *Deriving the characterization set*

5.5.3 *Identification sets*

5.6. The W-Method

5.6.1 *Assumptions*

5.6.2 *Maximum number of states*

5.6.3 *Computation of the transition cover set*

5.6.4 *Constructing Z*

5.6.5 *Deriving a test set*

5.6.6 *Testing using the W-method*

5.6.7 *The error detection process*

5.7. The Partial W-Method

5.7.1 Testing using the W_p-method for m = n

5.7.2 Testing using the W_p-method for m > n

5.8. The Uio-sequence Method

5.8.1 Assumptions

5.8.2 UIO sequences

5.8.3 Core and non-core behavior

5.8.4 Generation of UIO sequences

5.8.5 Explanation of gen-ueno

5.8.6 Distinguishing signatures

5.8.7 Test generation

5.8.8 Test optimization

5.8.9 Fault detection

5.9. Automata Theoretic Versus Control-Flow Based Techniques

5.9.1 n-switch-cover

5.9.2 Comparing automata theoretic methods

5.10. Tools

Summary

Exercises

6. Test Generation from Combinatorial Designs

6.1. Combinatorial Designs

6.1.1 Test configuration and test set

6.1.2 Modeling the input and configuration spaces

6.2. A Combinatorial Test Design Process

6.3. Fault Model

6.3.1 Fault vectors

6.4. Latin Squares

6.5. Mutually Orthogonal Latin Squares

6.6. Pairwise Design: Binary Factors

6.7. Pairwise Design: Multi-Valued Factors

6.7.1 Shortcomings of using MOLS for test design

6.8. Orthogonal Arrays

6.8.1 Mixed-level orthogonal arrays

6.9. Covering And Mixed-level Covering Arrays

6.9.1 Mixed-level covering arrays

6.10. Arrays of Strength > 2

6.11. Generating Covering Arrays

6.12. Tools

Summary

Exercises

PART III: TEST ADEQUACY ASSESSMENT AND ENHANCEMENT

7. Test Adequacy Assessment Using Control Flow and Data Flow

7.1. Test Adequacy: Basics

7.1.1 What is test adequacy?

7.1.2 Measurement of test adequacy

7.1.3 Test enhancement using measurements of adequacy

7.1.4 Infeasibility and test adequacy

7.1.5 Error detection and test enhancement

7.1.6 Single and multiple executions

7.2. Adequacy Criteria Based on Control Flow

7.2.1 Statement and block coverage

7.2.2 Conditions and decisions

7.2.3 Decision coverage

7.2.4 Condition coverage

7.2.5 Condition/decision coverage

7.2.6 Multiple condition coverage

7.2.7 Linear code sequence and jump (LCSAJ) coverage

7.2.8 Modified condition/decision coverage

7.2.9 MC/DC adequate tests for compound conditions

7.2.10 Definition of MC/DC coverage

7.2.11 Minimal MC/DC tests

7.2.12 Error detection and MC/DC adequacy

7.2.13 Short-circuit evaluation and infeasibility

7.2.14 Basis path coverage

7.2.15 Tracing test cases to requirements

7.3. Concepts from Data Flow

7.3.1 Definitions and uses

7.3.2 C-use and p-use

7.3.3 Global and local definitions and uses

7.3.4 Data flow graph

7.3.5 Def-clear paths

7.3.6 Def-use pairs

7.3.7 Def-use chains

7.3.8 A little optimization

7.3.9 Data contexts and ordered data contexts

7.4. Adequacy Criteria Based on Data Flow

7.4.1 c-use coverage

7.4.2 p-use coverage

7.4.3 All-uses coverage

7.4.4 k-dr chain coverage

7.4.5 Using the k-dr chain coverage

7.4.6 Infeasible c- and p-uses

7.4.7 Context coverage

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

Summary

Exercises

8. Test Adequacy Assessment Using Program Mutation

8.1. Introduction

8.2. Mutation and Mutants

8.2.1 First-order and higher order mutants

8.2.2 Syntax and semantics of mutants

8.2.3 Strong and weak mutations

8.2.4 Why mutate?

8.3. Test Assessment Using Mutation

8.3.1 A procedure for test adequacy assessment

8.3.2 Alternate procedures for test adequacy assessment

8.3.3 “Distinguished” versus “killed” mutants

8.3.4 Conditions for distinguishing a mutant

8.4. Mutation Operators

8.4.1 Operator types

8.4.2 Language dependence of mutation operators

8.5. Design of Mutation Operators

8.5.1 Goodness criteria for mutation operators

8.5.2 Guidelines

8.6. Founding Principles of Mutation Testing

8.6.1 The competent programmer hypothesis

8.6.2 The coupling effect

8.7. Equivalent Mutants

8.8. Fault Detection Using Mutation

8.9. Types of Mutants

8.10. Mutation Operators for C

8.10.1 What is not mutated?

8.10.2 Linearization

8.10.3 Execution sequence

8.10.4 Effect of an execution sequence

8.10.5 Global and local identifier sets

8.10.6 Global and local reference sets

8.10.7 Mutating program constants

8.10.8 Mutating operators

8.10.9 Binary operator mutations

8.10.10 Mutating statements

8.10.11 Mutating program variables

8.10.12 Structure Reference Replacement

8.11. Mutation Operators for Java

8.11.1 Traditional mutation operators

8.11.2 Inheritance

8.11.3 Polymorphism and dynamic binding

8.11.4 Method overloading

8.11.5 Java specific mutation operators

8.12. Comparison of Mutation Operators

8.13. Mutation Testing within Budget

8.13.1 Prioritizing functions to be mutated

8.13.2 Selecting a subset of mutation operators

8.14. Case and Program Testing

8.15. Tools

Summary

Exercises

PART IV: PHASES OF TESTING

9. Test Selection, Minimization, and Prioritization for Regression Testing

9.1. What is Regression Testing?

9.2. Regression Test Process

9.2.1 Revalidation, selection, minimization, and prioritization

9.2.2 Test setup

9.2.3 Test sequencing

9.2.4 Test execution

9.2.5 Output comparison

9.3. Regression Test Selection: the Problem

9.4. Selecting Regression Tests

9.4.1 Test all

9.4.2 Random selection

9.4.3 Selecting modification traversing tests

9.4.4 Test minimization

9.4.5 Test prioritization

9.5. Test Selection Using Execution Trace

9.5.1 Obtaining the execution trace

9.5.2 Selecting regression tests

9.5.3 Handling function calls

9.5.4 Handling changes in declarations

9.6. Test Selection Using Dynamic Slicing

9.6.1 Dynamic slicing

9.6.2 Computation of dynamic slices

9.6.3 Selecting tests

9.6.4 Potential dependence

9.6.5 Computing the relevant slice

9.6.6 Addition and deletion of statements

9.6.7 Identifying variables for slicing

9.6.8 Reduced dynamic dependence graph

9.7. Scalability of Test Selection Algorithms

9.8. Test Minimization

9.8.1 The set cover problem

9.8.2 A procedure for test minimization

9.9. Test Prioritization

9.10. Tools

Summary

Exercises

10. Unit Testing

10.1. Introduction

10.2. Context

10.3. Test Design

10.4. Using JUnit

10.5. Stubs and Mocks

10.5.1 *Using mock objects*

10.6. Tools

Summary

Exercises

11. Integration Testing

11.1. Introduction

11.2. Integration Errors

11.3. Dependence

11.3.1 *Class relationships: static*

11.3.2 *Class relationships: dynamic*

11.3.3 *Class firewalls*

11.3.4 *Precise and imprecise relationships*

11.4. OO versus Non-OO Programs

11.5. Integration Hierarchy

11.5.1 Choosing an integration strategy

11.5.2 Comparing integration strategies

11.5.3 Specific stubs and retesting

11.6. Finding a Near-optimal Test Order

11.6.1 The TD method

11.6.2 The TJM method

11.6.3 The BLW method

11.6.4 Comparison of TD, TJM, and the BLW methods

11.6.5 Which test order algorithm to select?

11.7. Test Generation

11.7.1 Data variety

11.7.2 Data constraints

11.8. Test Assessment

11.9. Tools

Summary

Exercises

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 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.

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.
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

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

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 researchers 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, W_p, 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	<i>Interim project presentations</i>	
	<i>Review, Midterm examination</i>	
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 literature
15, 16	<i>Review, Final project presentations</i>	
17	<i>Final examination</i>	

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 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.

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
	<i>Review, Midterm examination</i>	

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	<i>Review, Research presentations</i>	
17	<i>Final examination</i>	

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 TM JUnit CodeTest	IBM Rational Freeware Freescale Semiconductor

	xSuds	Telcordia Technologies
Defect tracking	Bugzilla FogBugz	Freeware Fog Creek Software
GUI testing	WebCorder jfcUnit	Crimson Solutions Freeware
Mutation testing	muJava Proteum	Professor Jeff Offutt offutt@ise.gmu.edu Professor Jose Maldonado jcmaldon@icmc.usp.br
Performance testing	Performance Tester JMeter	IBM Rational™ Apache, for Java
Regression testing	Eggplant xSuds	Redstone Software Telcordia Technologies
Test management	ClearQuest™ TestManager	IBM Rational™ IBM Rational™

<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

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.

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. 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 <i>foreign</i> cars. Heard: He has a garage for repairing <i>falling</i> 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: <i>waple malnut</i> , intent: <i>maple walnut</i> . Spoken: <i>We need a new refrigerator</i> , intent: <i>We need a new washing machine</i> .
Sports	Incorrect call by the referee in a tennis match.
Writing	Written: What kind of <i>pans</i> did you use ? Intent: What kind of <i>pants</i> did you use ?

Errors are a part of our daily life.

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 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 occurred. 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*.

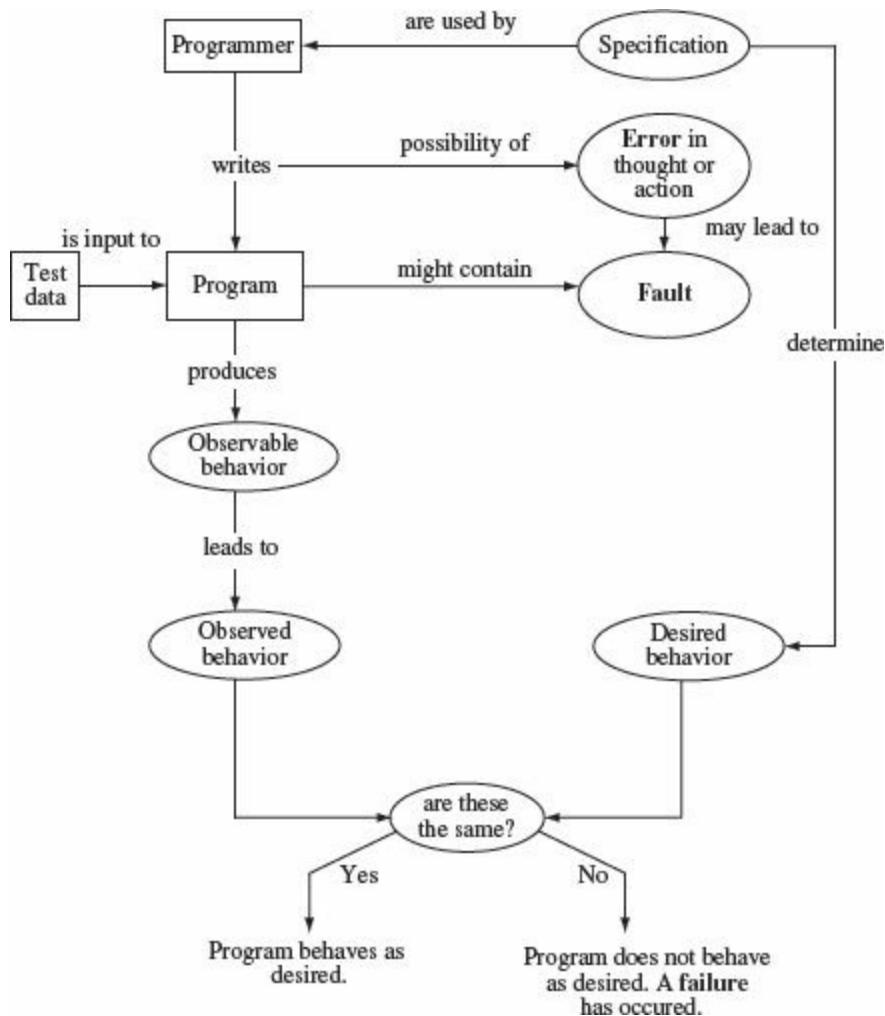


Figure 1.1 Errors, faults, and failures in the process of programming and testing.

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 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.

A developer is also a tester and vice-versa.

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

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.

Correctness and reliability are two dynamic attributes of software. Reliability can be considered as a statistical measure of correctness.

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.

$$\begin{aligned}[2 \ u \ u \ u] &\rightarrow [3 \ 3 \ 15 \ u] \rightarrow [4 \ 3 \ 15 \ 15] \rightarrow [5 \ 3 \ 15 \ 15] \\ &\rightarrow [8 \ 3 \ 15 \ 15] \rightarrow [9 \ 3 \ 15 \ 15]\end{aligned}$$

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 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.

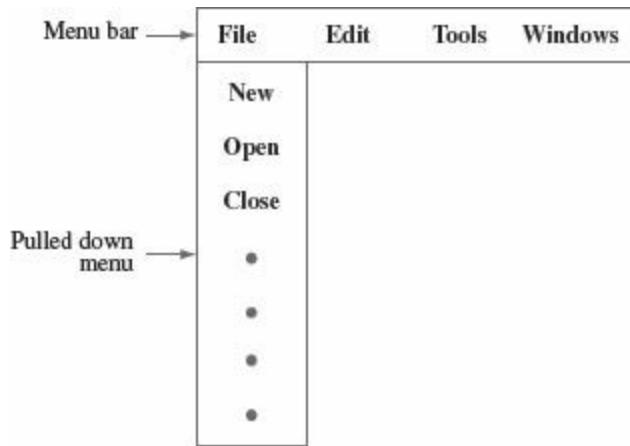


Figure 1.2 Menu bar displaying four menu items when application myapp is started.

[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.

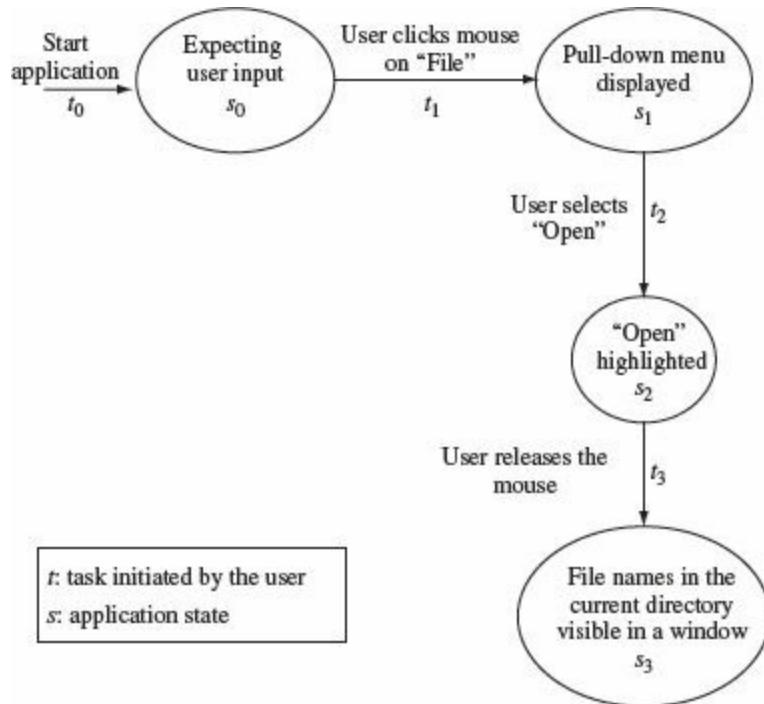


Figure 1.3 A state sequence for `myapp` showing how the application is expected to behave when the user selects the `open` option under the `File` menu item.

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

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.

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.

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

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.

Program correctness is established via mathematical proofs. Program reliability is established via testing. However, neither approach is foolproof.

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.

Testing a program for completeness does not ensure its correctness.

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.

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.”

Reliability is a statistical attribute. According to one view, it is the probability of failure free execution of a program under given conditions.

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

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*.

An operational profile is a statistical summary of how a program would be used in practice.

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.

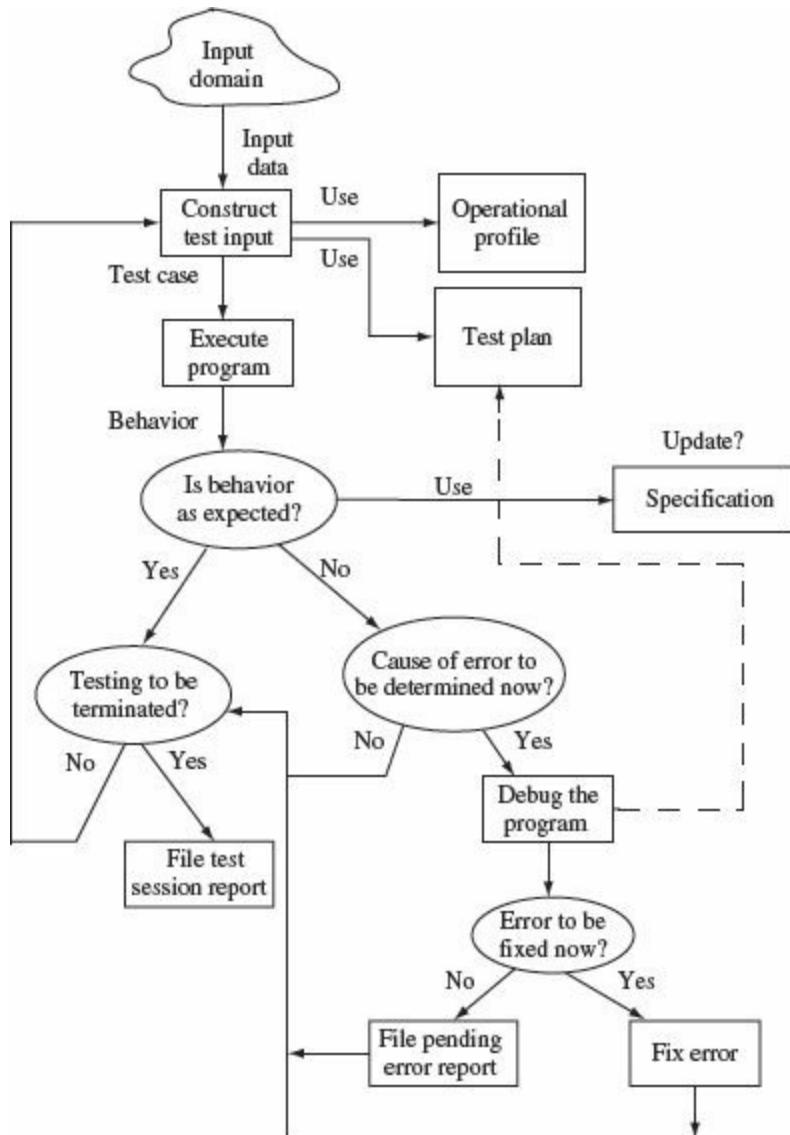


Figure 1.4 A test and debug cycle.

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](#).

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.

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. 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.

A test case is a pair of input data and the corresponding program output; a test set is a collection of test cases.

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.>  
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".>  
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, 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.

A test harness is an aid to testing a program.

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.

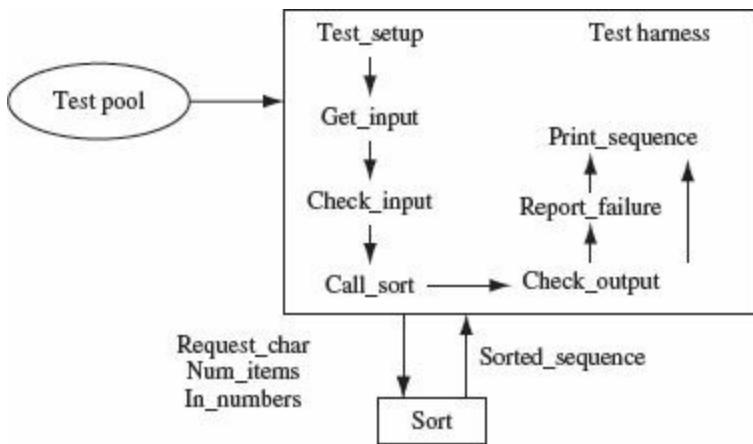


Figure 1.6 A simple test harness to test the `sort` program.

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 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.

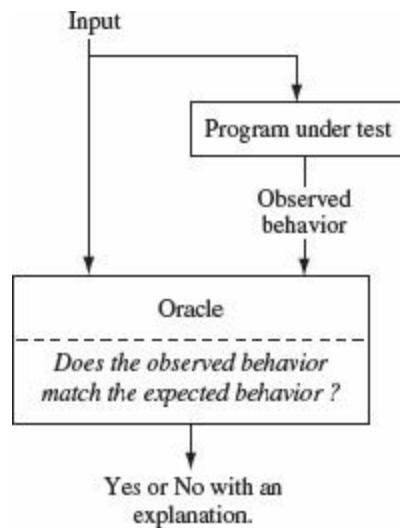


Figure 1.7 Relationship between the program under test and the oracle. The output from an oracle can be binary such as “Yes” or “No” or more complex such as an explanation as to why the oracle finds the observed behavior to be same or different from the expected behavior.

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.

An oracle is something that checks the behavior of a program. An oracle could itself be a program or a human being.

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.

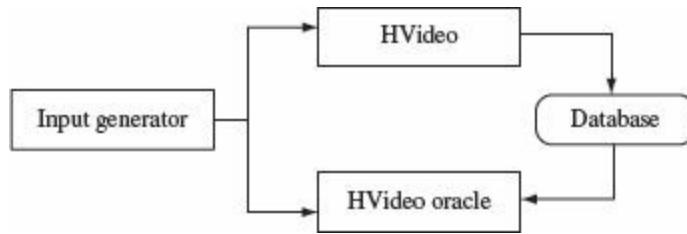


Figure 1.8 Relationship between an input generator, HVideo, and its oracle.

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 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.

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.

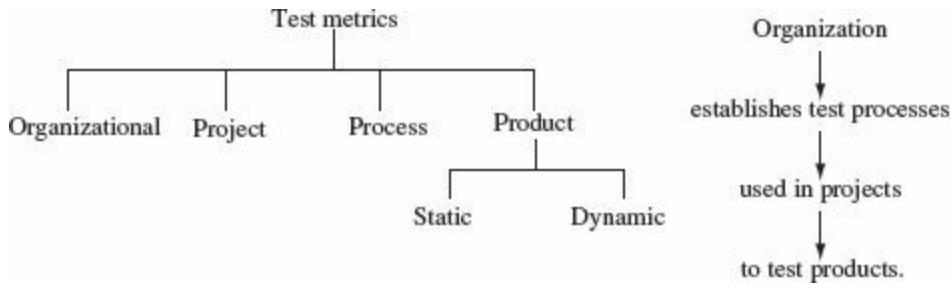


Figure 1.9 Types of metrics used in software testing and their relationships.

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.

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 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-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.

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.

The purpose of a process metric is to assess the “goodness” of a process.

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:

$$V(G) = E - N + 2p$$

The cyclomatic complexity is a measure of program complexity; higher values imply higher complexity. This is a product metric.

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](#).

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 coupling	class under consideration.
Response set	Measures the number of classes to which a given class is coupled.
Number of children	Set of all methods that can be invoked, directly and indirectly, when a message is sent to object O . Number of immediate descendants of a class in the class hierarchy.

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.

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.

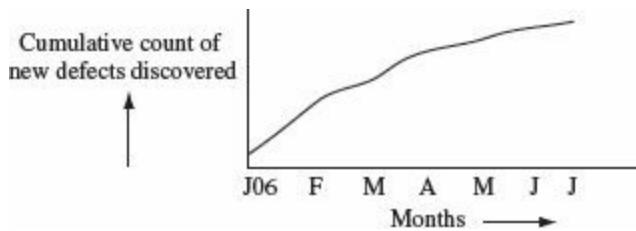


Figure 1.10 A sample plot of cumulative count of defects found over seven consecutive months in a software project.

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.

Product metrics could be classified as static or dynamic. Computing a dynamic metric will likely require program execution.

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 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 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.

Test domains differ between hardware and software tests. However, mutation testing of software is motivated by concepts in hardware testing.

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.

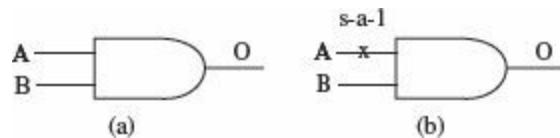


Figure 1.11 (a) A 2-input NAND gate, (b) A NAND gate with a stuck-at 1 fault in input A.

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

Stuck-at fault models are common in hardware.

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 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

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.

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 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 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 management is a practice used in several organizations. Several methods, e.g., Orthogonal Defect Classification, are available to categorize defects.

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 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.

Strategies for test generation abound. Very broadly these are classified as either black box or white box.

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](#).

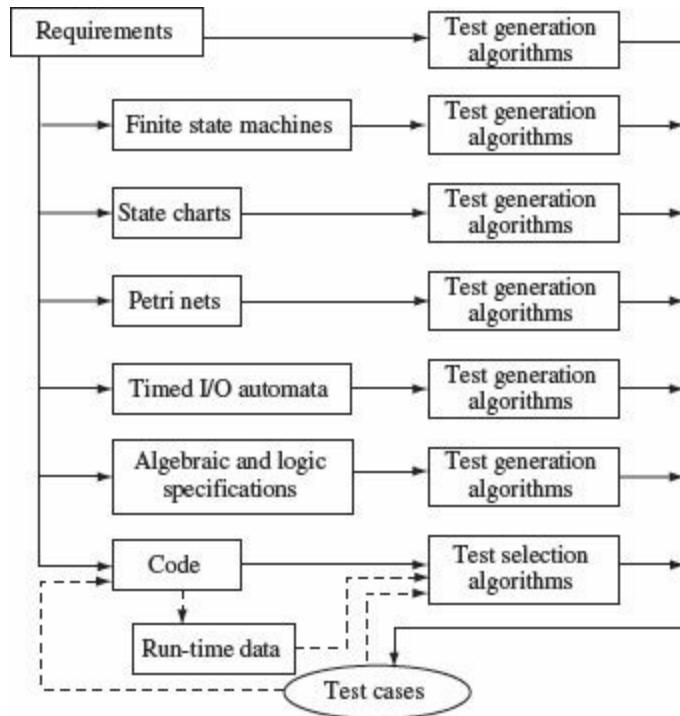


Figure 1.12 Requirements, models, and test generation algorithms.

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.

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 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 does not require code execution, dynamic testing does.

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.

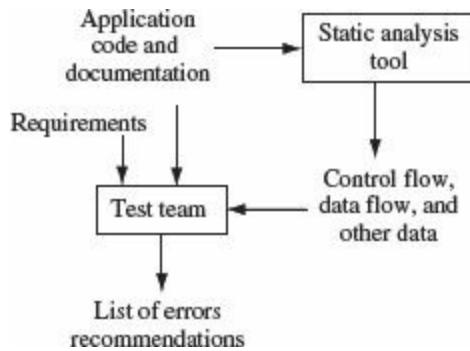


Figure 1.13 Elements of static testing.

1.11.1 Walkthroughs

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.

Code walkthrough, also known as peer code review, is used to review code and may be considered as a static testing technique.

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.

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.

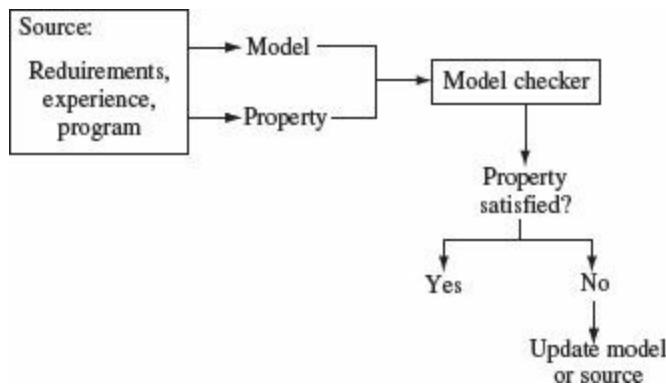


Figure 1.14 Elements of model checking.

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 counterexample 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.

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, pairwise testing could be used to design tests for an entire system or for a component.

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

Test techniques that fall under mapping C1 are more generic than others. Each technique under C1 is potentially applicable to meet the 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: 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.

Interface testing is a form of black-box testing where the focus is on testing a program via its interfaces.

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.

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

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 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.

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.

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.

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.”

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 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: 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.

Robustness testing aims to assess the robustness of an application against unintended, or invalid, inputs.

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: 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.

Stress testing tests an application for behavior under stress such as a large number of inputs or requests to a web server.

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: 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.

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.

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*.

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

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 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.

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.

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 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.

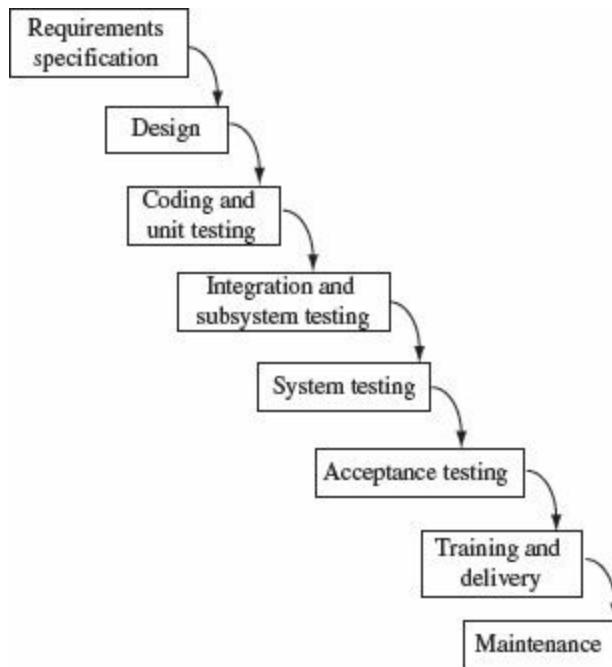


Figure 1.15 Testing in the waterfall model. Arrows indicate the “flow” of documents from one to the next. For example, design documents are input to the coding phase. The waterfall nature of the flow led to the name of this model.

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.

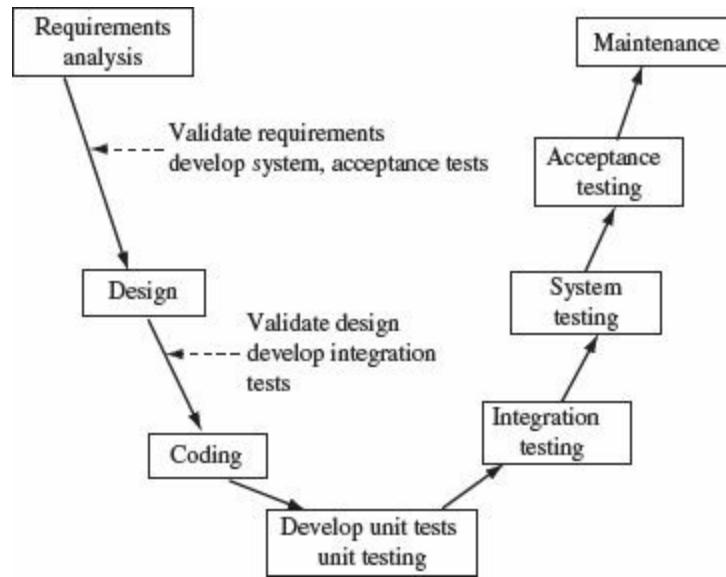


Figure 1.16 Testing in the V model.

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 a prototype evolves into an application. In spiral testing the sophistication of test activities increases with the stages of an evolving prototype.

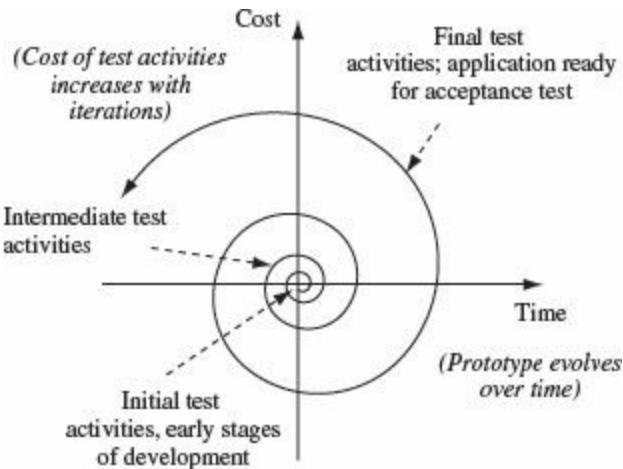


Figure 1.17 A visual representation of spiral testing. Test activities evolve over time and with the prototype. In the final iteration, the application is available for system and acceptance testing.

Spiral testing is generally used in an incremental process development model and leads to testing in all phases of the application development.

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.

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.

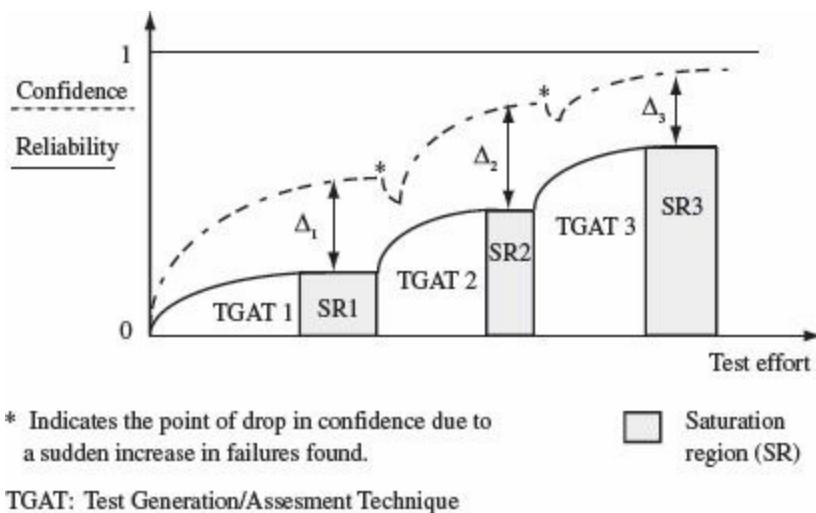


Figure 1.18 The saturation effect observed during the testing of complex software systems.

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.

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.

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 $T_2 \subset T_3$. 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 $T_1=T_2$ or that $T_2=T_3$. 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

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 \neg 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 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”

A *relational expression* is an expression of the form $e_1 \text{ relop } e_2$ 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	$p \wedge q \vee \neg r \wedge s$	p, q, r , and s are Boolean variables.
Non-singular Boolean expression	$p \wedge q \vee \neg r \wedge p$	

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 $p \wedge q \vee \neg r \wedge s$ can be written as $pq + \neg rs$.

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 $pq + \neg rs$ is a sum of two product terms pq and $\neg rs$. 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 $p \wedge q \vee \neg r \wedge p$.

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, \dots, e_k .

$$E = e_1 \text{ bop } e_2 \text{ bop } \dots \text{ } e_{k-1} \text{ bop } e_k.$$

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 singular if each variable in the expression occurs exactly once.

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.

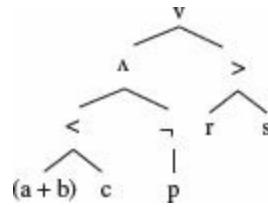


Figure 2.1 An abstract syntax tree for compound predicate $(a + b < c) \wedge (\neg p) \vee (r > s)$.

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. 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 control flow graph (CFG) captures the flow of control in a program. It consists of basic blocks and directed arcs connecting the blocks.

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

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.

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.

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 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.

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.

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:

$$\begin{aligned}N &= \{\text{Start}, 1, 2, 3, 4, 5, 6, 7, 8, 9, \text{End}\} \\E &= \{(\text{Start}, 1), (1, 2), (1, 3), (2, 4), (3, 4), (4, 5), (5, 6), \\&\quad (6, 5), (5, 7), (7, 8), (7, 9), (8, 9), (9, \text{End})\}\end{aligned}$$

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.

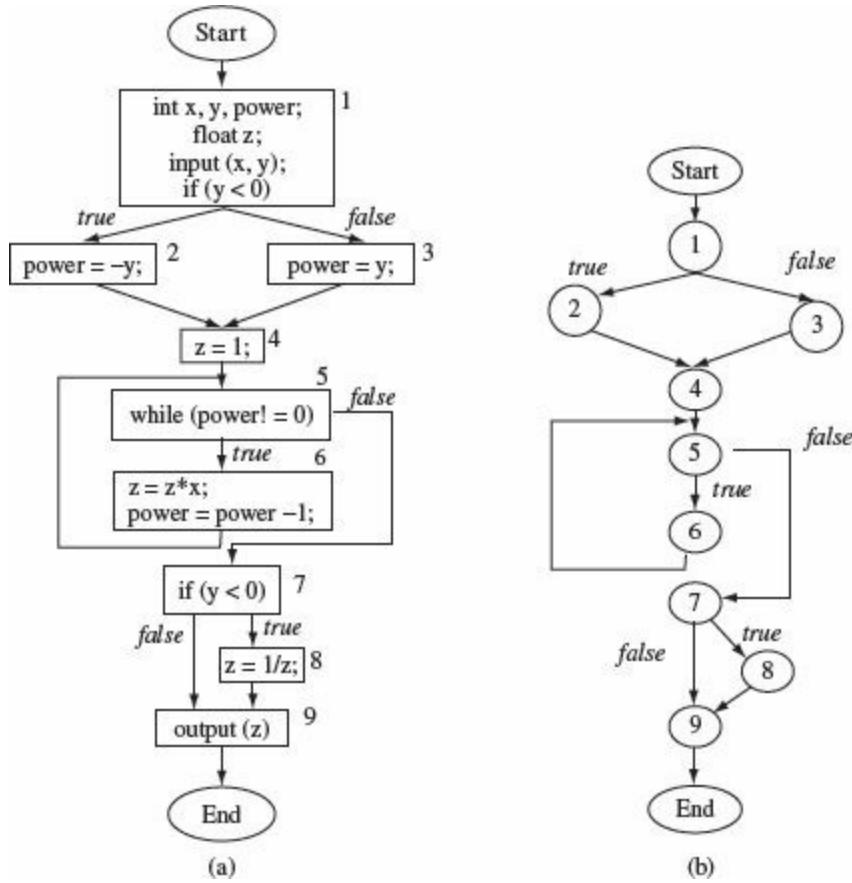


Figure 2.2 Flow graph representations of Program P2.1. (a) Statements in each block are shown. (b) Statements within a block are omitted.

2.2.3 Paths

Consider a flow graph $G = (N, E)$. A sequence of k edges, $k > 0$, (e_1, e_2, \dots, 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 $n_q = n_r$.

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.

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 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 $\text{succ}(n)$ and $\text{pred}(n)$, 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 $p = \{n_1, n_2, \dots, n_t\}$ and $s = \{i_1, i_2, \dots, i_u\}$, s is a subpath of p if for some $1 \leq j \leq t$ and $j + u - 1 \leq t$, $i_1 = n_j$, $i_2 = n_{j+1}, \dots, i_u = n_{j+u-1}$. 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:

$$p_1 = (\text{Start}, 1, 2, 4, 5, 6, 5, 7, 8, 9, \text{End})$$

$$p_2 = (\text{Start}, 1, 3, 4, 5, 6, 5, 7, 9, \text{End})$$

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 .

$$V_{p1} = [1101011111011]$$

$$V_{p1} = [1010111110101]$$

$(4, 5, 6)$ is a subpath of p_1 . $(\text{Start}, 1, 2, 4, 5)$ 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](#).

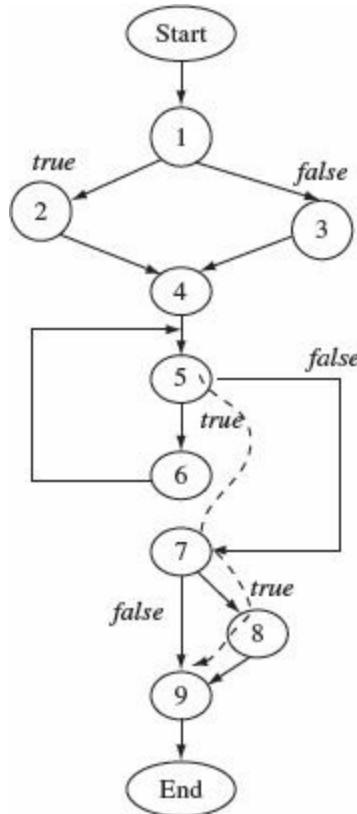


Figure 2.3 Flow graph representation of Program [P2.1](#). A complete path is shown using bold edges and a sub-path using a dashed line.

$$p_3 = (5, 7, 8, 9)$$

$$p_4 = (6, 5, 7, 9, \text{End})$$

The next two paths of lengths 10 and 7 are complete but infeasible.

$$p_5 = (\text{Start}, 1, 3, 4, 5, 6, 5, 7, 8, 9, \text{End})$$

$$p_6 = (\text{Start}, 1, 2, 4, 5, 7, 9, \text{End})$$

Finally, the next two paths are invalid as they do not satisfy the sequence condition stated earlier.

$p_7 = (\text{Start}, 1, 2, 4, 8, 9, \text{End})$

$p_8 = (\text{Start}, 1, 2, 4, 7, 9, \text{End})$

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 descendants of node 5. We also have $\text{succ}(5) = \{5, 6, 7, 8, 9, \text{End}\}$ and $\text{pred}(5) = \{\text{start}, 1, 2, 3, 4, 5, 6\}$. Note that in the presence of loops, a node can be its own ancestor or descendent.

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.

Each additional condition in a program adds at least one path.

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 ( $C_1$ ) { . . . }  
    .  
    .  
    .  
    if ( $C_2$ ) { . . . }  
    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
```

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.

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.

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

```
1 begin
2 int num, product;
3 bool done;
4 product=1;
5 input(done);
6 while (!done) {
7     input(num);
8     product=product*num;
9     input(done);
10 }
11 output(product);
12 end
```

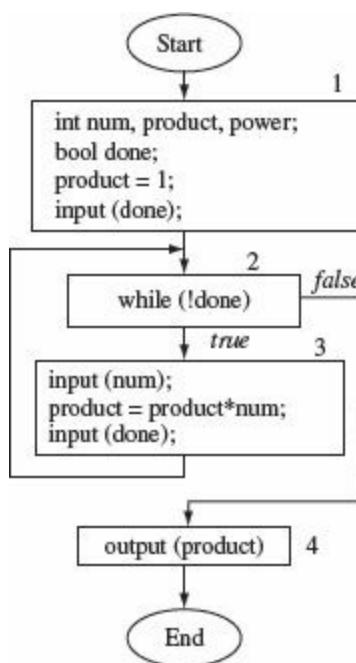


Figure 2.4 Flow graph of Program P2.2. Numbers 1 through 4 indicate the four basic blocks in P2.2.

As shown in [Figure 2.4](#), Program P2.2 contains four basic blocks and one condition that guards the body of `while`. $(\text{Start}, 1, 2, 4, \text{End})$ 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 $(\text{Start}, 1, 2, 3, 2, 4, \text{End})$. When there are two input integers to be multiplied then the path traversed is $(\text{Start}, 1, 2, 3, 2, 3, 2, 4, \text{End})$.

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, \dots, V_n denote the path vectors of the basis paths. Then V_p can be represented as the following linear combination.

$V_p = a_1 V_1 + a_2 V_2 \dots + a_n V_n$, where a_1, a_2, \dots, 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](#).

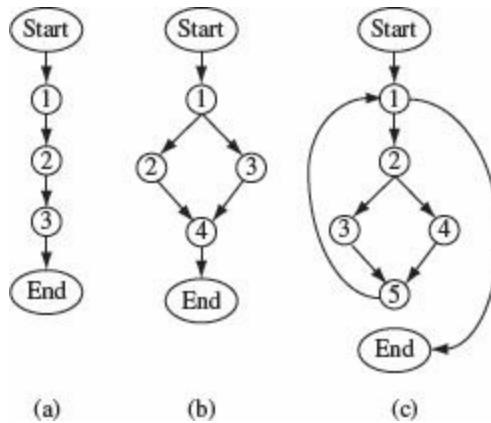


Figure 2.5 Sample control flow graphs to illustrate basis paths.

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 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 1 0 1 1 1]

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 .

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 x_1 and y_1 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 $x \leq y$: $p_{12} = (1, 2, 3, 4, 5, 6, 9, 10, 5, 11)$

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.

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.

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.

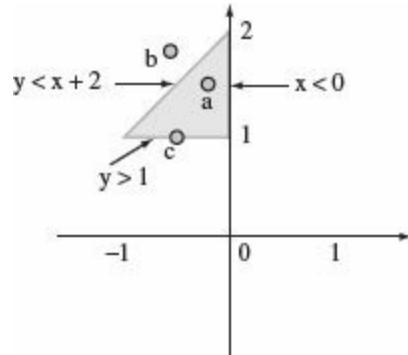


Figure 2.6 Pictorial representation of a path domain corresponding to the condition $y < x + 2 \wedge y > 1 \wedge x < 0$. The shaded triangular area represents the path domain which is a subdomain of the entire program domain. Three sample points labeled a, b, and c, are shown. Point a lies inside the path domain, b lies outside, and c lies on one of the borders.

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 form of a set of m path conditions in n inputs x_1, x_2, \dots, x_n can be expressed as follows.

$$\begin{array}{lll} f1(x_1, x_2, \dots, x_n) & relop & 0 \\ f2(x_1, x_2, \dots, x_n) & relop & 0 \\ \vdots & & \\ fm(x_1, x_2, \dots, x_n) & relop & 0 \end{array}$$

In case all functions in the above set of conditions are linear, the path conditions can be written as follows.

$$\begin{array}{lll} a_{11}x_1 + a_{12}x_2, \dots + a_{1n}x_n + c_1 & relop & 0 \\ a_{21}x_1 + a_{22}x_2, \dots + a_{2n}x_n + c_2 & relop & 0 \\ \vdots & & \\ a_{m1}x_1 + a_{m2}x_2, \dots + a_{mn}x_n + c_m & relop & 0 \end{array}$$

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.

$$\begin{array}{lll} -x + y - 2 & < & 0 \\ -y + 1 & < & 0 \\ x & < & 0 \end{array}$$

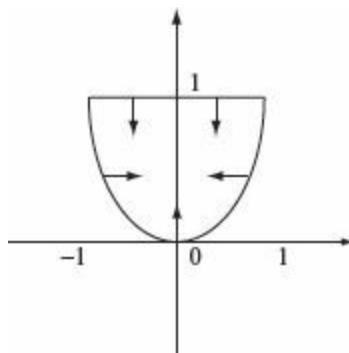


Figure 2.7 Non-linear path domain corresponding to the path condition $y > x^2 \wedge y < 1$. The arrows indicate the closed boundary of the parabolic domain.

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.

$$\begin{array}{ll} y - x^2 > 0 \\ y < 1 \end{array}$$

[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.

```
1 int x, n;  
2 input (x, n);
```

```
3  inputArray(n, A);
4  . . .
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](#).)

2.2.6 Domain and computation errors

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 is said to exist in a program if for some input an incorrect path leads to an incorrect output.

A computation error is said to exist in a program if for some input a correct path leads to an incorrect output.

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 `acce1` is used at line 42 in module `updateAcce1` but where is it defined ?” The author might respond as “`acce1` is defined in module `computeAcce1`”. 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 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.

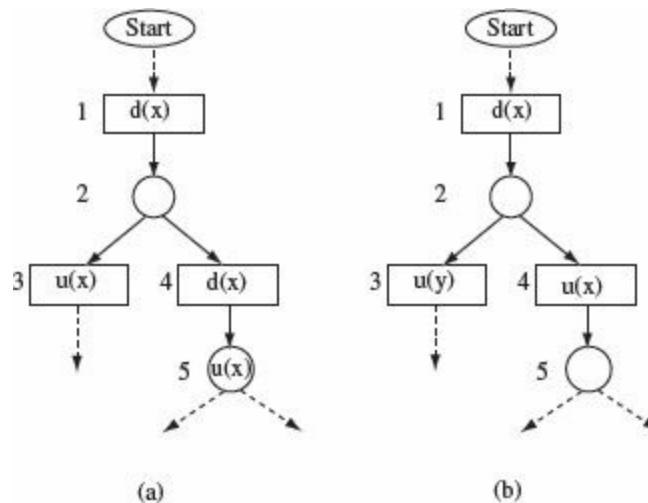


Figure 2.8 Partial CFGs annotated with data flow information. $d(x)$ and $u(x)$, respectively, imply the definition and use of variable x in a block, (a) CFG that indicates a possible data flow error, (b) CFG with a data flow error.

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

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.

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.

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 $t_1 : \langle x = 2, y = 3 \rangle$. 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 $t_2 : \langle x = 1, y = 0 \rangle$ 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 portion 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

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](#)).

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.

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 $\text{dom}(n, m)$ 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 $\text{pdom}(n, m)$ when n post-dominates m . When $n \neq m$, we refer to these as strict dominator and strict post-dominator relations. $\text{Dom}(n)$ and $\text{PDom}(n)$ denote, respectively the sets of dominators and post-dominators of node n .

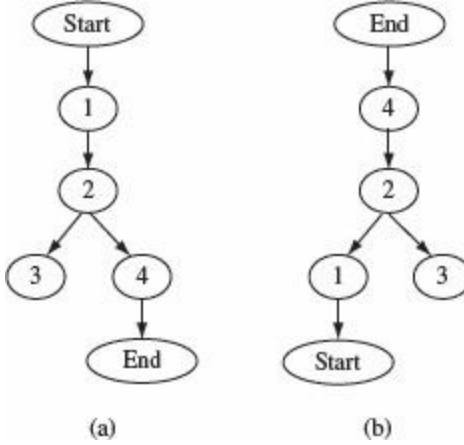


Figure 2.9 (a) Dominator and (b) post-dominator trees derived from the flow graph in [Figure 2.4](#).

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 $idom(4, end)$. Similarly, from the post-dominator tree, we obtain $ipdom(4, 2)$ and $ipdom(end, 4)$.

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 $\text{Dom}(4)$ is $\{2, 1, \text{Start}\}$.

$\text{Dom}(4)$ 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 $\{4, \text{End}\}$.

2.5 Program Dependence Graph

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.

A program dependence graph captures data and control dependence relations in a program.

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-zero 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 corresponds 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.

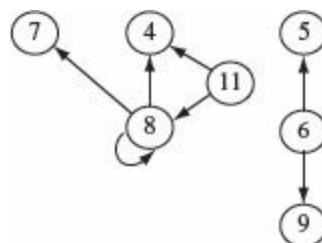


Figure 2.10 Data dependence graph for Program P2.2. Node numbers correspond to line numbers. Declarations have been omitted.

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 .

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 dependence 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](#)).

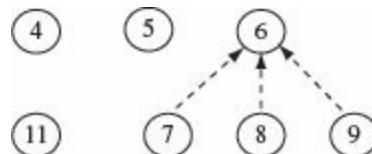


Figure 2.11 Control dependence graph for [Program P2.2](#).

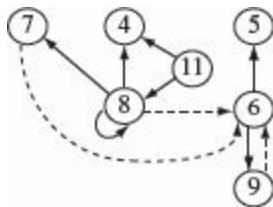


Figure 2.12 Program dependence graph for [P2.2](#).

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.

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.

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 $Y = \{\text{dog, cat, horse, lion}\}$ that consists of four symbols “dog”, “cat”, “horse”, and “lion”.

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 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 $\{\text{dog, cat, horse, lion}\}$. 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 $|\text{dogcatdog}| = 3$. A string of length 0, also known as an *empty string*, is denoted by \in .

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 $s \bullet \in = s$ and $\in \bullet s = s$.

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 concatenation as $L_1 \bullet L_2$ that denotes the set L defined as:

$$L = L_1 \bullet L_2 = \{x \bullet y \mid x \in L_1, y \in L_2\}$$

The following sets are finite languages over the binary alphabet $\{0, 1\}$.

\emptyset : The empty set

$\{\in\}$: 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 $r_1 \cdot r_2$ is a regular expression that denotes the set $L_1 \cdot L_2$.
- 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 $\{\in\} \cup L^+$.
- 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://eclipsecfg.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.grammotech.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.

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](#).

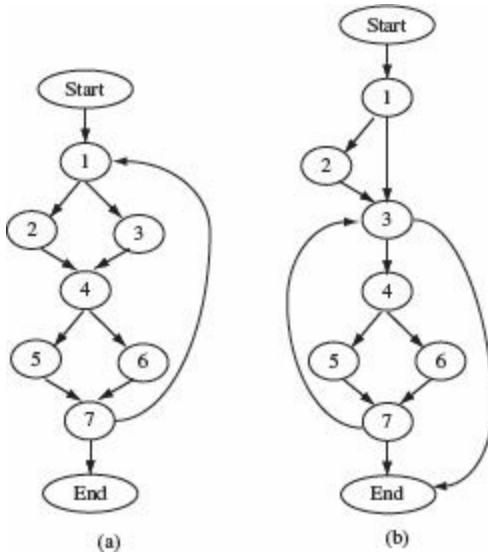


Figure 2.13 Control flow graphs for [Exercise 2.7](#).

2.8 Let $\text{pred}(n)$ 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:

$$\text{Dom}(\text{Start}) = \{\text{Start}\}$$

$$\text{Dom}(n) = \{n\} \cup \left\{ \bigcap_{p \in \text{pred}(n)} \text{Dom}(p) \right\}$$

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.

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 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.

Requirements may be formal or informal. In either case, they serve as a key source of tests.

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 techniques, listed in the figure, are used to select a relatively small number of test cases from a usually large input domain.

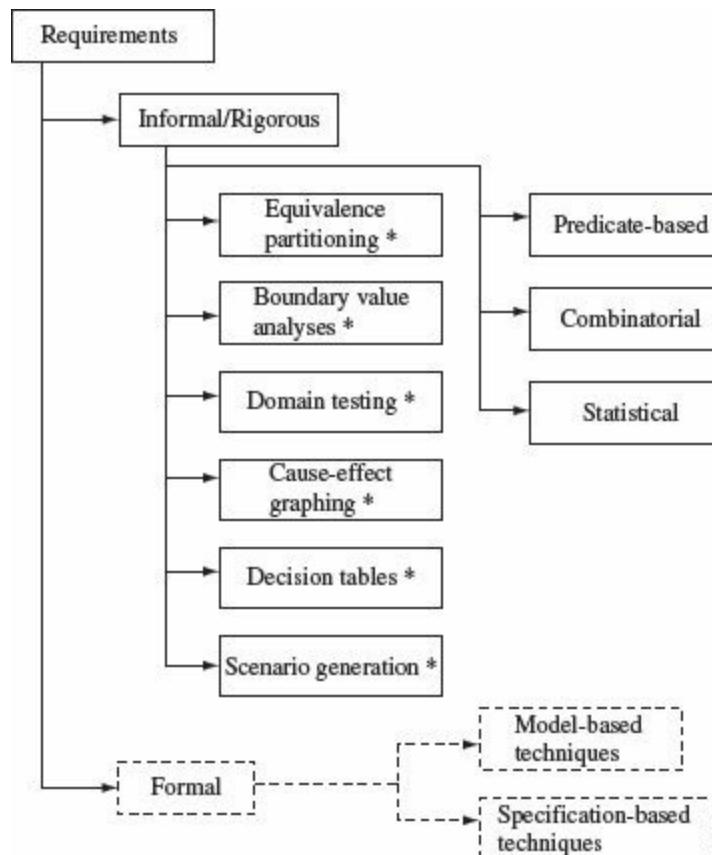


Figure 3.1 Techniques listed here, and marked with an asterisk, for test selection from informal and rigorously specified requirements, are introduced in this chapter. Techniques from formally specified requirements using graphical models and logic-based languages are discussed in other chapters.

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 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 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.

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

$$S = \sum_{i=0}^N v^i,$$

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: name is 20 characters long; each character belongs to the string; 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: hoursWorked varies from 0 to 60.

int;

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

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*.

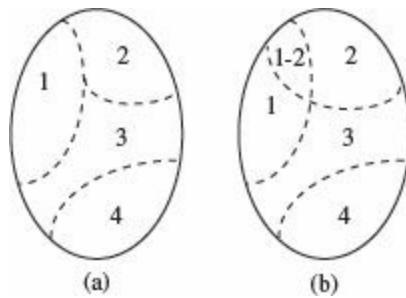


Figure 3.2 Four equivalence classes created from an input domain. (a) Equivalence classes together constitute a partition as they are disjoint. (b) Equivalence classes are not disjoint and do not form a partition. 1-2 indicates the region where subsets 1 and 2 overlap. This region contains test inputs that belong to subsets 1 and 2.

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.

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.

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*

The entire set of inputs to any application can be divided into at 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.

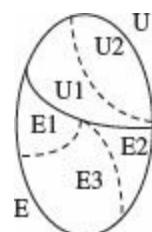


Figure 3.3 Set of inputs partitioned into two regions one containing expected (E), or legal, and the other containing unexpected (U), or illegal, inputs. Regions E and U are further subdivided based on the expected behavior of the application under test.

Representative tests, one from each region, is targeted at exposing faults that cause the application to behave incorrectly in their respective regions.

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.

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 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.

$$\text{addList} : L \rightarrow \mathbb{Z}$$

where 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.

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.

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:

$$\text{addList} : L \rightarrow \mathbb{Z} \cup \{\text{error}\}$$

Relations that help a tester partition the input domain of a program are usually of the kind

$$R: I \rightarrow I,$$

where I denotes the input domain. Relation R is *on* the input domain. It defines an *equivalence class* that is a subset of 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`.

$$pFound : I \rightarrow I$$

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 *I* of *gPrice*. Together, these equivalence classes form a partition of the input domain *I* because $pF \cup pNF = I$ 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 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 I . However, if `pTest` provides a graphical user interface to select a printer, then 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.

$$ci : I \rightarrow I$$

$$cl : I \rightarrow I$$

$$cm : I \rightarrow I$$

$$invP : I \rightarrow I$$

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.

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 constraints imposed by the underlying hardware and software of the machine on which the application is being tested.

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\}\downarrow, \{75\}\uparrow, \{92\}\downarrow}
		$area : float;$ $area \geq 0$	{\{-1.0\}\downarrow, \{15.52\}\uparrow}
		$age : int;$ $0 \leq age \leq 120$	{\{-1\}\downarrow, \{56\}\uparrow, \{132\}\downarrow}
		$letter : char;$	{\{J\}\uparrow, \{3\}\downarrow}
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.	$fname : string;$	{\{\epsilon\}\downarrow, \{Sue\}\uparrow, \{Sue2\}\downarrow, \{Too Long a name\}\downarrow}
		$vname : string;$	{\{\epsilon\}\downarrow, \{shape\}\uparrow, \{address1\}\uparrow, \{Long variable\}\downarrow}

[†]Symbols following each equivalence class: \downarrow : Representative from an equivalence class containing illegal inputs. \uparrow : Representative from an equivalence class containing legal inputs.

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.	<i>auto_color</i> ∈ {red, blue, green}	{ {red} }↑, { {blue} }↑, { {green} }↑ }
		<i>up</i> : 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: <code>int[] aName = new int[3];</code>	{ { [] } }↓, { { [-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.

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.

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.

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 for *letter* have been determined based on the assumption that *letter* must be one of 26 letters A through Z.

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.

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.

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 *Sue29* 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 *address1*, which contains a digit, is a legal value for *vname*, *Sue2*, 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.

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.

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.

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

```
1 struct R
2 {
3     string fName; // First name.
4     string lName; // Last name.
5     string cTitle [200]; // Course titles.
6     char grades [200]; // Letter grades corresponding to
course titles.
7 }
```

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 I as the set product of the input variables and define a relation on 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 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.

$$\begin{array}{lll} E1: & x < 3 & E2: & 3 \leq x \leq 7 & E3: & x > 7 \\ E4: & y < 5 & E5: & 5 \leq y \leq 9 & E6: & y > 9 \end{array}$$

[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 .

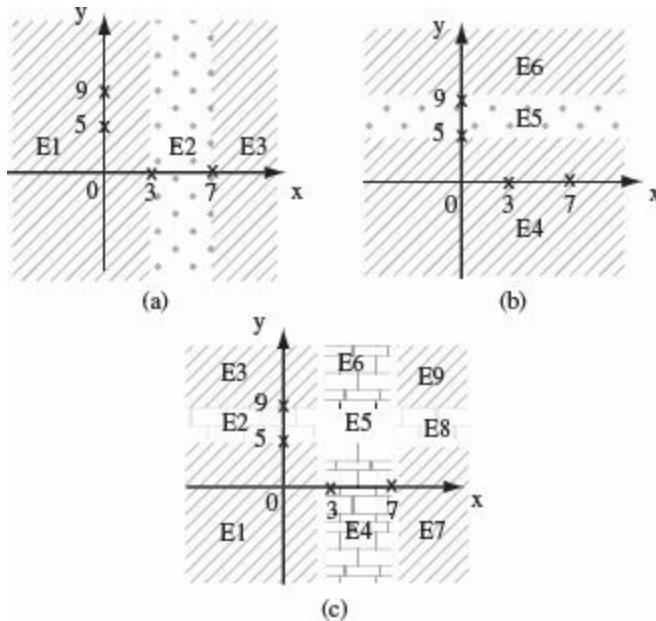


Figure 3.4 Geometric representation of equivalence classes derived using unidimensional partitioning based on x and y as in (a) and (b), respectively, and using multidimensional partitioning as in (c).

$$\begin{array}{lll}
 \text{E1: } x < 3, y < 5 & \text{E2: } x < 3, 5 \leq y \leq 9 & \text{E3: } x < 3, y > 9 \\
 \text{E4: } 3 \leq x \leq 7, y < 5 & \text{E5: } 3 \leq x \leq 7, 5 \leq y \leq 9 & \text{E6: } 3 \leq x \leq 7, y > 9 \\
 \text{E7: } x > 7, y < 5 & \text{E8: } x > 7, 5 \leq y \leq 9 & \text{E9: } x > 7, y > 9
 \end{array}$$

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 unidimensional partitioning. On the other hand, the number of equivalence classes created using multidimensional partitioning increases exponentially in the number of input variables. (See [Exercises 3.8, 3.9, 3.10, and 3.11](#).)

3.3.5 A systematic procedure

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.

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.

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](#).

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.

An infeasible equivalence class contains inputs that cannot be applied to the program under test.

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.

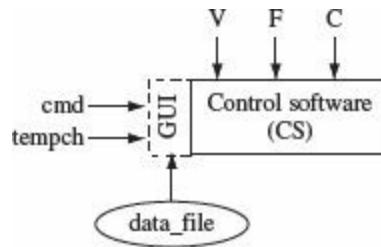


Figure 3.5 Inputs to the boiler control software. V and F are environment variables. Values of cmd (command) and $tempch$ (temperature change) are input via the GUI or a data file depending on V . F specifies the data file.

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 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}

While deriving equivalence classes, it is useful to consider illegal values as possible inputs.

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 I , contains S . Sample values in 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 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.

$\{(GUI, f_valid, temp, t_invalid)\}$ $\{(GUI, f_invalid, temp, t_invalid)\}$	<i>The GUI does not allow invalid values of temperature change to be input.</i>
--	---

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.*

$\{(file, f_valid, temp, t_valid \cup t_invalid)\}$ *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

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 $pTest$.

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.

$$T = \{\text{HP cp1700, Canon Laser Shot LBP 3200,} \\ \text{Epson Stylus Photo RX600, My Printer} \\ \}$$

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	$\{(GUI, f_valid, temp, t_valid)\}$ $\{(GUI, f_valid, temp, t_valid)\}$	$(GUI, a_file, temp, 10)$
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	$\{(GUI, f_invalid\ temp, t_valid)\}$ $\{(GUI, f_invalid\ temp, t_valid)\}$	$(GUI, no_file, temp, -10)$
E9	$\{(GUI, _, cancel, NA)\}$	$(GUI, a_file, cancel, -5)$
E10	$\{(GUI, _, cancel, NA)\}$	$(GUI, no_file, cancel, -5)$
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	$\{(file, f_valid, temp, t_valid)\}$ $\{(file, f_valid, temp, t_valid)\}$ $\{(file, f_valid, temp, t_invalid)\}$	$(file, a_file, temp, -25)$
E16	$\{(file, f_valid, temp, NA)\}$	$(file, a_file, shut, 10)$
E17	$\{(file, f_invalid, NA, NA)\}$	$(file, no_file, shut, 10)$

E18	$\{(undefined, _, NA, NA)\}$	$(undefined, no_file, shut, 10)$
-----	-------------------------------	-----------------------------------

$_$: Don't care. NA: Input not allowed.

a_file : File exists. no_file : File does not exist.

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.

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.

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.

$$T = \{ (GUI, a_file, temp, -10), (GUI, no_file, temp, -5), \\ (file, a_file, temp, 5), (file, a_file, cancel, 10), \\ (undefined, a_file, shut, -10) \\ \}$$

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

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.

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.

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\(b\)](#).

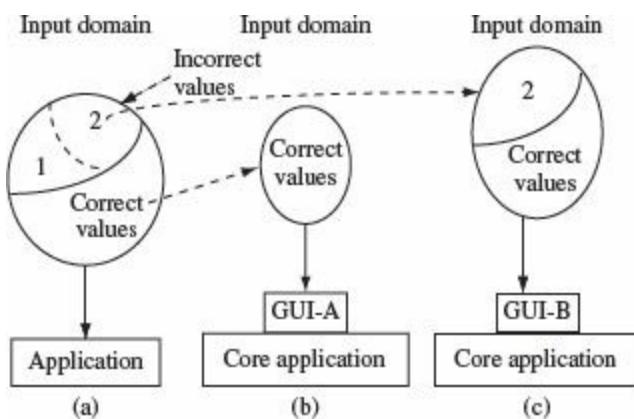


Figure 3.6 Restriction of the input domain through careful design of the GUI. Partitioning of the input domain into equivalence classes must account for the presence of GUI as shown in (b) and (c). GUI-A protects all variables against incorrect input while GUI-B does allow the possibility of incorrect input for some variables.

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 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.

A GUI serving as an interface to a program could reduce its input domain.

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 $x \leq 0$ 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.

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.

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.

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 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.

$$\begin{aligned} T = & \{ t_1 : (code = 98, qty = 0), \\ & t_2 : (code = 99, qty = 1), \\ & t_3 : (code = 100, qty = 2), \\ & t_4 : (code = 998, qty = 99), \\ & t_5 : (code = 999, qty = 100), \\ & t_6 : (code = 1000, qty = 101) \end{aligned}$$

}

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.

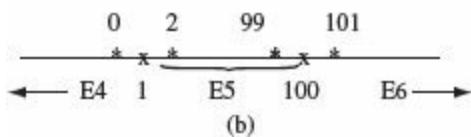
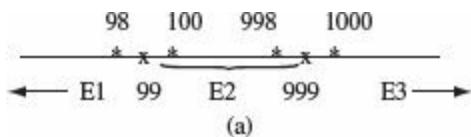


Figure 3.7 Equivalence classes and boundaries for variables (a) code and (b) qty in [Example 3.11](#). Values at and near the boundary are listed and marked with an “x” and “*” respectively.

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*.

```
1  public void fP(int code, qty)
2  {
3      if (code<99 || code>999)
4          {display_error(''Invalid code''); return;}
5      // Validity check for qty is missing!
6      // Begin processing code and qty.
7      :
8  }
```

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.

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.

$$t_7 = (\text{code} = 98, \text{qty} = 45)$$

$$t_8 = (\text{code} = 1000, \text{qty} = 45)$$

$$t_9 = (\text{code} = 250, \text{qty} = 0)$$

$$t_{10} = (\text{code} = 250, \text{qty} = 101)$$

One could improve the chances of detecting a missing code by separating the tests for legal and illegal values of different input variables.

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:

$s = \text{"Laughter"}$ and $txt = \text{"Laughter is good for the heart."}$

Based on the six equivalence classes and two boundaries derived, we obtain the following set T of three test inputs for textSearch .

$$T = \{ t_1 : (s = \epsilon, \text{txt} = \text{"Laughter is good for the heart."}), \\ t_2 : (s = \text{"Laughter"}, \text{txt} = \epsilon), \\ t_3 : (s = \text{"good for"}, \text{txt} = \text{"Laughter is good for the heart."}), \\ t_4 : (s = \text{"Laughter"}, \text{txt} = \text{"Laughter is good for the heart."}) \}$$

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 .

$$t_5 : (s = \text{"heart."}, \text{txt} = \text{"Laughter is good for the heart."})$$

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.

- $t_6 : (s = \text{"aughter"}, txt = \text{"Laughter is good for the heart."}),$
 $t_7 : (s = \text{"heart"}, txt = \text{"Laughter is good for the heart."}),$
 $t_8 : (s = \text{"gLaughter"}, txt = \text{"Laughter is good for the heart."}),$
 $t_9 : (s = \text{"heart.d"}, txt = \text{"Laughter is good for the heart."}).$

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.

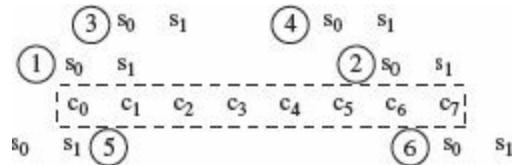


Figure 3.8 c_0 through c_7 are the eight characters in txt starting at the leftmost position 0. s_0s_1 is the input string s . Shown here is the positioning of s with respect to txt at the two boundaries, labeled 1 and 2, and at four points near the boundary, labeled 3 through 6.

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 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.

The category-partition method, and an associated tool, help ease the implementation of equivalence partitioning and boundary value analysis.

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.

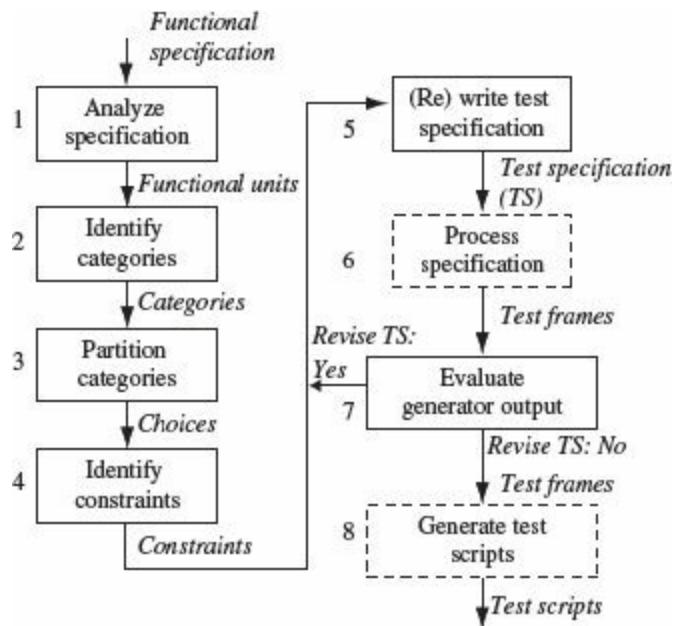


Figure 3.9 Steps in the generation of tests using the category-partition method. Tasks in solid boxes are performed manually and generally difficult to automate. Dashed boxes indicate tasks that can be automated.

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.

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

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.

Prior to generating test scripts, one needs to identify independently testable units of a program.

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.

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.

A characteristic of an input or an environment variable is considered a category.

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

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](#).

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 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]
<i>qty</i>	
valid quantity	[if ordinary-grocery]
invalid quantity	[error]
<i>weight</i>	
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 into 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 $\sum_{i=0}^N v^i$.

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

$$cl : I \rightarrow \{\text{yes}, \text{no}\}$$

cl maps an input domain I of $pTest$ in [Example 3.4](#) to the set $\{\text{yes}, \text{no}\}$. 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: and $T_1 = T_2$, $T_1 \subset T_2$, $T_1 \subseteq T_2$, $T_1 \supset T_2$, $T_1 \supseteq T_2$, and $T_1 \neq T_2$? (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 $0 \leq \text{age} \leq 120$. *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 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.

```

1  public double incorrectHierons(double w, double e, double cw,
2      double ce){
3      final double mw=40; // Should be 30.
4      final double discount=0.2;
5      if(w>=mw){
6          return(w*cw+discount*ce*e);
7      }else{
8          return(w*cw+ce*e);
9      }

```

- (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\(b\)](#) 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 $y < x + 1$.

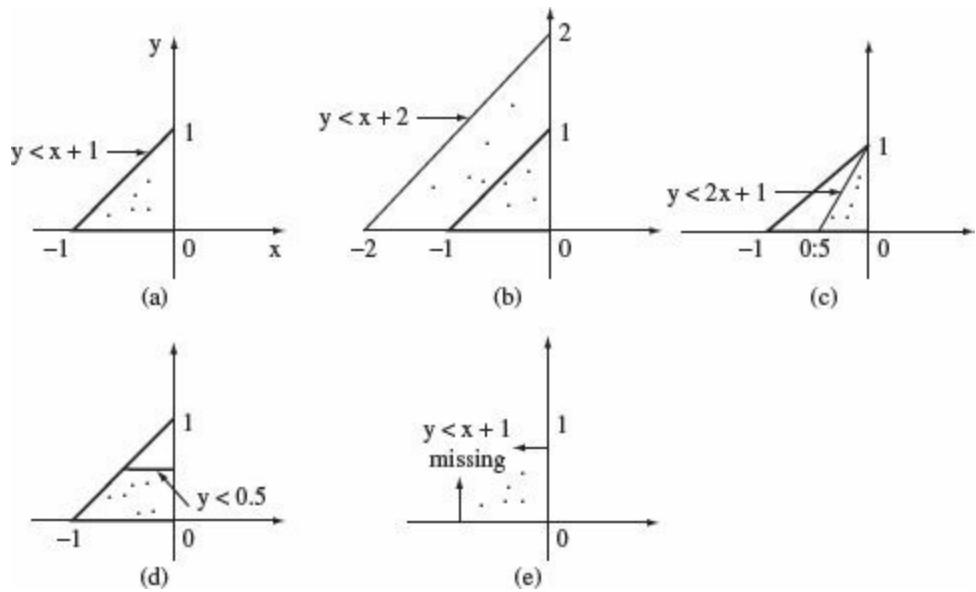


Figure 4.1 Types of domain errors. (a) Correct boundary corresponding to the path condition $y < x + 1 \wedge y > 0 \wedge x < 0$. (b) and (c) Boundary shifted due to error in $y < x + 1$. (d) Extra border due to the inclusion of $y < 0.5$ in the path condition. (e) Missing border due to missing $y < x + 1$. The dots indicate points inside the boundary.

4.2.2 Border shifts

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.

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 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 (\neq) 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.

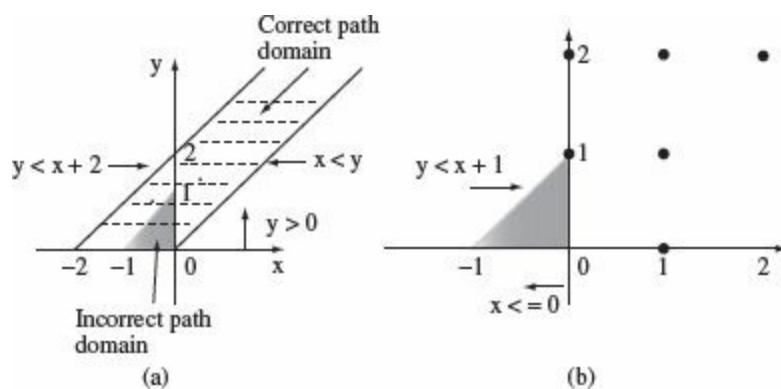


Figure 4.2 (a) Path domains corresponding to the incorrect and correct path conditions in [Example 4.1](#). (b) Incorrect path domain, scaled up, and the ON-OFF points.

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.

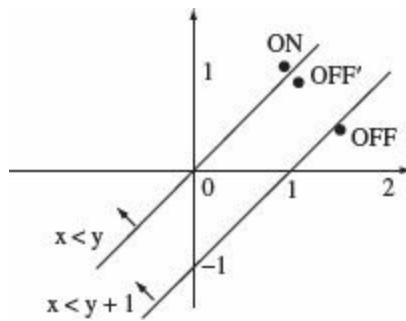


Figure 4.3 Path domains corresponding $x < y$ (incorrect) and $x < y + 1$ (correct). ON and OFF points shown do not detect the error. Note that the correct border lies between the incorrect border and the OFF point. OFF' point does detect the error as it lies between the incorrect and the correct borders (see [Example 4.3](#).)

Now suppose we select the OFF point very near the incorrect border, as it should have been. Thus, for example, OFF: ($x = 1, y = 0.999$). 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.

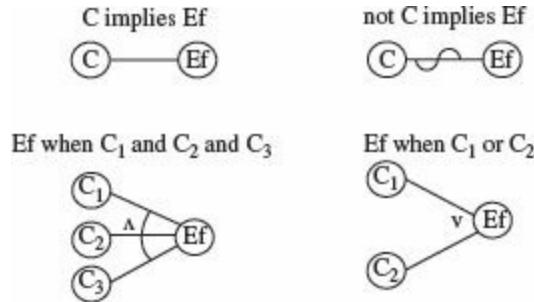


Figure 4.4 Basic elements of a cause-effect graph: implication, not (\sim), and (\wedge), or (\vee). C , C_1 , C_2 , and C_3 denote causes and Ef denotes effect. An arc is used, for example in the and relationship, to group three or more causes.

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 ($C_1 \wedge C_2 \wedge C_3$) then Ef ;
Ef when C_1 or C_2 :	if ($C_1 \vee C_2$) 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.

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.

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.

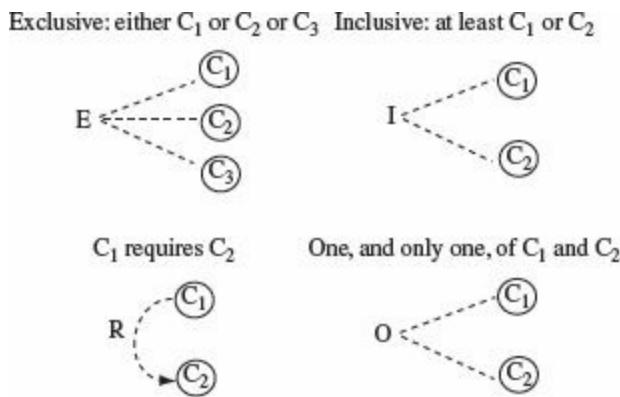


Figure 4.5 Constraints amongst causes (E, I, O, and R).

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

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.

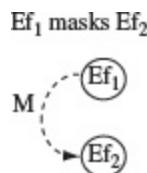


Figure 4.6 The masking constraint amongst effects.

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.

$$\left\{ \begin{array}{l} \text{CPU 1} \\ \text{CPU 2} \\ \text{CPU 3} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{PR 1} \\ \text{PR 2} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{M 20} \\ \text{M 23} \\ \text{M 30} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{RAM 256} \\ \text{RAM 512} \\ \text{RAM 1G} \end{array} \right\}$$

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 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, 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.

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 .

Once the conditions (causes) and the expected outcome (effects) have been identified, the relationship amongst them can be expressed using a cause-effect graph.

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:

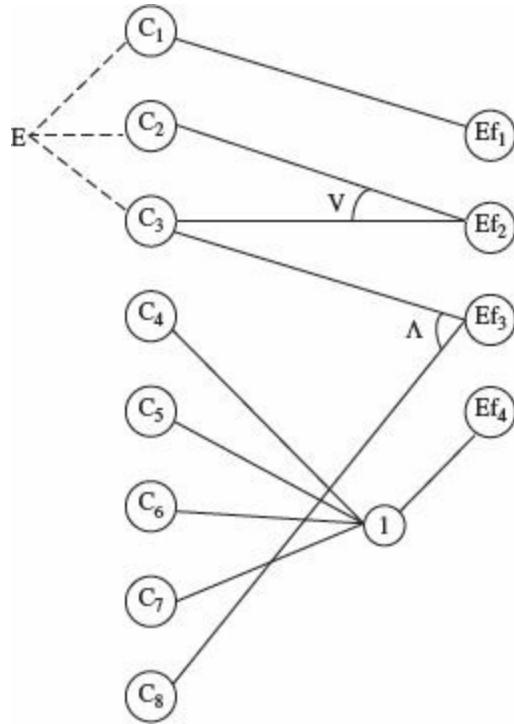


Figure 4.8 Cause-effect graph for the web-based computer sale application. C_1, C_2 , and C_3 denote the purchase of, respectively, CPU 1, CPU 2, and CPU 3. C_4 and C_5 denote the purchase of printers PR 1 and PR 2, respectively. C_6, C_7 , and C_8 denote the purchase of monitors M 20, M 23, and M 30, respectively.

$$Ef_1 = C_1$$

$$Ef_2 = C_2 \vee C_3$$

$$Ef_3 = C_3 \wedge C_8$$

$$Ef_4 = C_4 \wedge C_5 \wedge C_6 \wedge C_7$$

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, \dots, C_p and effects Ef_1, Ef_2, \dots, 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 $e = Ef_i$.

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, \dots, 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, \dots, 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, \dots, 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

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.

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 d, 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](#), $e = Ef_1$. 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

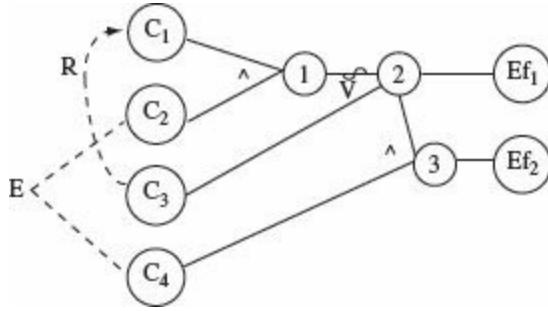


Figure 4.9 A cause-effect graph to illustrate procedure CEGDT.

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 $e = Ef_2$. 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.

$$\begin{array}{cccc} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{array}$$

Appending the corresponding values of Ef_1 and Ef_2 to each of the above combinations, we obtain the following three vectors.

$$\begin{array}{ccccccc} V_1 & 1 & 0 & 1 & 1 & 1 & 1 \\ V_2 & 1 & 0 & 0 & 1 & 1 & 1 \\ V_3 & 0 & 0 & 0 & 1 & 1 & 1 \end{array}$$

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

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 .

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.

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.

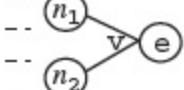
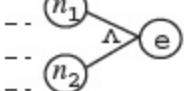
While the application of heuristics will likely reduce the number of tests, it might also reduce the error detection effectiveness of the generated test suite.

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](#).

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 $n_1 = n_2 = 1$.

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.

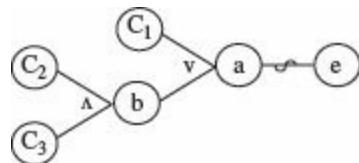


Figure 4.10 Cause-effect graph for [Example 4.7](#).

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 to the inputs to node a listed in the rightmost column are separated by horizontal lines.

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.

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	

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( $c_1 \vee c_2$ ) print("Error");
```

Application of a heuristic may or may not save on tests.

Now consider the following erroneous implementation of the same requirement.

```
if( $c_1 \vee \neg c_2$ ) 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:

$C: x < 99.$

Tests are generated from the decision table obtain from a cause-effect graph. Each combination of conditions may generate more than one test case.

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.

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.

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.

$$p_r: (\text{printer_status}=\text{ON}) \wedge (\text{printer_tray} = \neg \text{empty})$$

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 $p(t) = \text{false}$. 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.

$(a < b) \wedge (c > d) \wedge e$	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).

The incorrect use of a Boolean operator is known as a *Boolean operator fault*.

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.

$(a == b) \vee (c > d) \wedge e$	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.

The incorrect use of a relational operator is known as a *relational operator fault*.

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 $e_1 \text{ relop}_1 e_2$ and an incorrect relational expression E_i to be $e_3 \text{ relop}_2 e_4$. 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.

- 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 $e_1 = e_2$.

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.

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 $ a - b = 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 $ a - (b + 1) \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 $ a - (b - 1) > 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 $t_1 : \langle a = 0, b = 0, c = 0 \rangle$ and $t_2 : \langle a = 0, b = 0, c = 1 \rangle$. 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

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.

A missing or extra Boolean variable leads to an incorrect path condition.

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 be 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 $< a = 1, c = 0, d = 1 > \epsilon$ 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.

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**.

A predicate constraint for a condition is considered infeasible if it cannot be satisfied by any value of the variables in the condition.

We will now define constraints for entire predicates that contain Boolean variables and relational expressions joined by one or more Boolean operators.

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*.

A predicate constraint for a compound predicate having n simple conditions is a sequence of $(n + 1)$ BR symbols, one for each simple condition.

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 .

$\langle b = \text{true}, r = 1, s = 1, u = 1, v = 0 \rangle$

Several other test cases also satisfy C for p_r . The following test case does not satisfy C for p_r .

$\langle b = \text{true}, r = 1, s = 1, u = 1, v = 2 \rangle$

as the last of the three elements in C is not satisfied for the corresponding component of p_r which is $u \geq v$.

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 $p_r(C) = \text{true}$ 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 $C \in S^t$, $p_r(C) = \text{true}$ and for each $C \in S^f$, $p_r(C) = \text{false}$.

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.

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**perator,” “**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 ; $S = \{(t, t), (t, f), (f, t)\}$. The following test set T satisfies constraint set S and the BOR testing criterion.

$$\begin{aligned} T = \{ &t_1 : \langle a = 1, b = 2, c = 1, d = 0 \rangle; \text{Satisfies } (t, t), \\ &t_2 : \langle a = 1, b = 2, c = 1, d = 2 \rangle; \text{Satisfies } (t, f), \\ &t_3 : \langle a = 1, b = 0, c = 1, d = 0 \rangle; \text{Satisfies } (f, t) \\ &\} \end{aligned}$$

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 check this by evaluating p_r , and all its variants created by inserting Boolean operator faults, against T .

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>

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.

$$A \times B = \{(a, b) \mid a \in A, b \in B\} \quad (4.1)$$

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 $A = \{t, =, >\}$ and $B = \{f, <\}$. Using the definitions of set product and the onto product, we get the following sets:

$$A \times B = \{(t, f), (t, <), (=, f), (=, <), (>, f), (>, <)\}$$

$$A \otimes B = \{(t, f), (=, <), (>, <)\}; \text{One possibility.}$$

$$A \otimes B = \{(t, <), (=, f), (>, <)\}; \text{Second possibility.}$$

$$A \otimes B = \{(t, f), (=, <), (>, f)\}; \text{Third possibility.}$$

$$A \otimes B = \{(t, <), (=, <), (>, f)\}; \text{Fourth possibility.}$$

$$A \otimes B = \{(t, <), (=, f), (>, f)\}; \text{Fifth possibility.}$$

$$A \otimes B = \{(t, f), (=, f), (>, <)\}; \text{Sixth possibility.}$$

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

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 .

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.

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 $AST(pr)$.

Procedure: BOR-CSET

Step 1 Label each leaf node N of $AST(p_r)$ with its constraint set $S(N)$ for each leaf $S_N = \{t, f\}$.

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*:

$$\begin{aligned} S_N^f &= S_{N_1}^f \otimes S_{N_2}^f \\ S_N^t &= (S_{N_1}^t \times \{f_2\}) \cup (\{f_1\} \times S_{N_2}^t) \\ \text{where } f_1 &\in S_{N_1}^f \text{ and } f_2 \in S_{N_2}^f \end{aligned}$$

2.2 N is an *AND-node*:

$$\begin{aligned} S_N^t &= S_{N_1}^t \otimes S_{N_2}^t \\ S_N^f &= (S_{N_1}^f \times \{t_2\}) \cup (\{t_1\} \times S_{N_2}^f) \\ \text{where } t_1 &\in S_{N_1}^t \text{ and } t_2 \in S_{N_2}^f \end{aligned}$$

2.3 N is *NOT-node*:

$$\begin{aligned} S_N^t &= S_{N_1}^f \\ S_N^f &= S_{N_1}^t \end{aligned}$$

Step 3 The constraint set for the root of $AST(p_r)$ is the desired
BOR constraint set for p_r .

End of Procedure BRO-CSET

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.

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\(a\)](#).

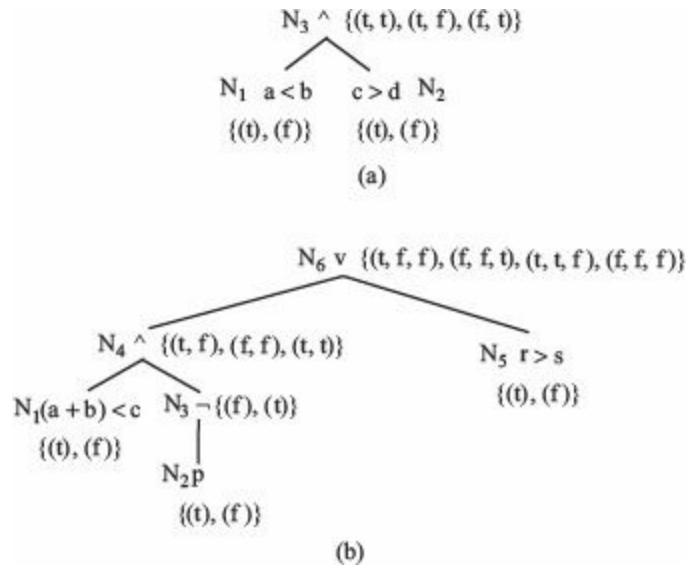


Figure 4.11 BOR constraint sets for predicates (a) $a < b \wedge c > d$ and (b) $(a + b) < c \wedge \neg p \vee (r > s)$. The constraint sets are listed next to each node. See the text for the separation of each constraint set into its true and false components.

N_1 and N_2 are the leaf nodes. The constraint sets for these two nodes are given below.

$$\begin{aligned} S_{N_1}^t &= \{t\}, & S_{N_1}^f &= \{f\} \\ S_{N_2}^t &= \{t\}, & S_{N_2}^f &= \{f\} \end{aligned}$$

Traversing $AST(p_1)$ bottom up, we compute the constraint set for non-leaf N_3 which is an AND-node.

$$\begin{aligned} S_{N_3}^t &= S_{N_1}^t \otimes S_{N_2}^t \\ &= \{t\} \otimes \{t\} \\ &= \{(t,t)\} \\ S_{N_3}^f &= (S_{N_1}^f \times \{t_2\}) \cup (\{t_1\} \times S_{N_2}^f) \\ &= (\{f\} \times \{t\}) \cup (\{t\} \times \{f\}) \\ &= \{(f,t), (t,f)\} \end{aligned}$$

Thus, we obtain $S_{N_3} = \{(t,t), (f,t), (t,f)\}$ 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.

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 $((a + b < c) \wedge (\neg p)) \vee (r > s)$.

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.

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:

$$\begin{aligned} S'_{N_3} &= S'_{N_2} = \{\text{f}\} \\ S^f_{N_3} &= S^f_{N_2} = \{\text{t}\} \end{aligned}$$

The following BOR constraint set is obtained by applying the rule for the AND-node.

$$\begin{aligned}
S_{N_4}^t &= S_{N_1}^t \otimes S_{N_3}^t \\
&= \{\mathbf{t}\} \otimes \{\mathbf{f}\} \\
&= \{(\mathbf{t}, \mathbf{f})\} \\
S_{N_4}^f &= (S_{N_1}^f \times \{t_{N_3}\}) \cup (\{t_{N_1}\} \times S_{N_3}^f) \\
&= (\{\mathbf{f}\} \times \{\mathbf{f}\}) \cup (\{\mathbf{t}\} \times \{\mathbf{t}\}) \\
&= \{(\mathbf{f}, \mathbf{f}), (\mathbf{t}, \mathbf{t})\} \\
S_{N_4} &= \{(\mathbf{t}, \mathbf{f}), (\mathbf{f}, \mathbf{f}), (\mathbf{t}, \mathbf{t})\}
\end{aligned}$$

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:

$$\begin{aligned}
S_{N_6}^f &= S_{N_4}^f \otimes S_{N_5}^f \\
&= \{(\mathbf{f}, \mathbf{f}), (\mathbf{t}, \mathbf{t})\} \otimes \{\mathbf{f}\} \\
&= \{(\mathbf{f}, \mathbf{f}, \mathbf{f}), (\mathbf{t}, \mathbf{t}, \mathbf{f})\} \\
S_{N_6}^t &= (S_{N_4}^t \times \{f_{N_5}\}) \cup (\{f_{N_4}\} \times S_{N_5}^t) \\
&= (\{(\mathbf{t}, \mathbf{f})\} \times \{\mathbf{f}\}) \cup (\{(\mathbf{f}, \mathbf{f})\} \times \{\mathbf{t}\}) \\
&= \{(\mathbf{t}, \mathbf{f}, \mathbf{f}), (\mathbf{f}, \mathbf{f}, \mathbf{t})\} \\
S_{N_6} &= \{(\mathbf{t}, \mathbf{f}, \mathbf{f}), (\mathbf{f}, \mathbf{f}, \mathbf{t}), (\mathbf{t}, \mathbf{t}, \mathbf{f}), (\mathbf{f}, \mathbf{f}, \mathbf{f})\}
\end{aligned}$$

Notice that we could select any one of the constraints (\mathbf{f}, \mathbf{f}) or (\mathbf{t}, \mathbf{t}) for f_{N_4} . Here we have arbitrarily selected (\mathbf{f}, \mathbf{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.

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	$< a = 1, b = 1, c = 3, p = \text{false}, r = 1, s = 2 >$
t_2	f	f	t	$< a = 1, b = 1, c = 1, p = \text{false}, r = 1, s = 0 >$
t_3	t	t	f	$< a = 1, b = 1, c = 3, p = \text{true}, r = 1, s = 1 >$
t_4	f	f	f	$< a = 1, b = 1, c = 0, p = \text{false}, r = 0, s = 0 >$

A test set for a predicate is derived from its BOR constraint set by selecting suitable values of the variables involved in the predicate.

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 $e_1 \text{ relop } e_2$ 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.

$$\begin{array}{lll}
relOp : > & S^r = \{(>)\} & S^f = \{(\text{=}), (<)\} \\
relOp : \geq & S^r = \{(>), (\text{=})\} & S^f = \{(<)\} \\
relOp := & S^r = \{(\text{=})\} & S^f = \{(<), (>)\} \\
relOp : < & S^r = \{(<)\} & S^f = \{(\text{=}), (>)\} \\
relOp : \leq & S^r = \{(<), (\text{=})\} & S^f = \{(>)\}
\end{array}$$

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.

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, $S_N = \{\text{t}, \text{f}\}$. For each leaf node that is a relational expression, $S_N = \{(>), (\text{=}), (<)\}$.

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_{N1} and S_{N2} 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

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.

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 , as shown in Figure 4.12, is derived using Step 2.3 of Procedure BOR-CSET.

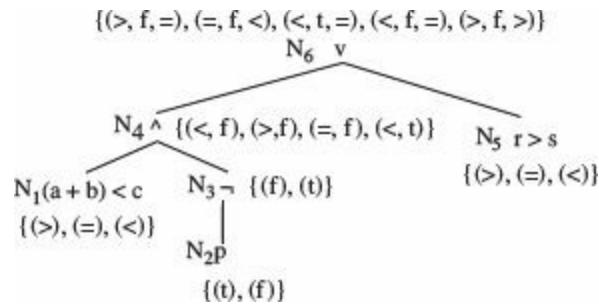


Figure 4.12 BRO constraint set for predicate $p_r = (a + b) < c \wedge \neg p \vee (r > s)$. The constraint sets are listed next to each node. See the text for the separation of each constraint set into its true and false components.

Next we construct the BRO constraint set for the AND node N_4 using Step 2.2.

$$\begin{aligned}
S_{N_4}^t &= S_{N_1}^t \otimes S_{N_3}^t \\
&= \{(<)\} \otimes \{\mathbf{f}\} \\
&= \{(<, \mathbf{f})\} \\
S_{N_4}^f &= (S_{N_1}^f \times \{t_{N_3}\}) \cup (\{t_{N_1}\} \times S_{N_3}^f) \\
&= \{(>), (=)\} \times \{\mathbf{f}\} \cup \{(<)\} \times \{\mathbf{t}\} \\
&= \{(>, \mathbf{f}), (=, \mathbf{f}), (<, \mathbf{t})\} \\
S_{N_4} &= \{(<, \mathbf{f}), (>, \mathbf{f}), (=, \mathbf{f}), (<, \mathbf{t})\}
\end{aligned}$$

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 .

$$\begin{aligned}
S_{N_6}^f &= S_{N_4}^f \otimes S_{N_5}^f \\
&= \{(>, \mathbf{f}), (=, \mathbf{f}), (<, \mathbf{t})\} \otimes \{(<), (=)\} \\
&= \{(>, \mathbf{f}, =), (=, \mathbf{f}, <), (<, \mathbf{t}, =)\} \\
S_{N_6}^t &= (S_{N_4}^t \times \{f_{N_5}\}) \cup (\{f_{N_4}\} \times S_{N_5}^t) \\
&= \{(<, \mathbf{f})\} \times \{(\mathbf{t})\} \cup \{(>, \mathbf{f})\} \times \{(>)\} \\
&= \{(<, \mathbf{f}, =), (>, \mathbf{f}, >)\} \\
S_{N_6} &= \{(>, \mathbf{f}, =), (=, \mathbf{f}, <), (<, \mathbf{t}, =), (<, \mathbf{f}, =), (>, \mathbf{f}, >)\}
\end{aligned}$$

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.

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	=	$\langle a = 1, b = 1, c = 1, p_r = \text{false}, r = 1, s = 1 \rangle$
t_2	=	f	<	$\langle a = 1, b = 0, c = 1, p_r = \text{false}, r = 1, s = 2 \rangle$
t_3	<	t	=	$\langle a = 1, b = 1, c = 3, p_r = \text{true}, r = 1, s = 1 \rangle$
t_4	<	f	=	$\langle a = 0, b = 2, c = 3, p_r = \text{false}, r = 0, s = 0 \rangle$
t_5	>	f	>	$\langle a = 1, b = 1, c = 0, p_r = \text{false}, r = 2, s = 0 \rangle$

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 $\{\text{t, f}\}$ 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 $e_1 \text{ relop } e_2$ 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”.

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, constraint S into its true and false components as follows:

<i>relop:></i>	$S' = \{(+\epsilon)\}$	$S' = \{(\text{=}), (-\epsilon)\}$
<i>relop:\geq</i>	$S' = \{(+\epsilon), (\text{=})\}$	$S' = \{(-\epsilon)\}$
<i>relop:=</i>	$S' = \{(\text{=})\}$	$S' = \{(+\epsilon), (-\epsilon)\}$
<i>relop:<</i>	$S' = \{(-\epsilon)\}$	$S' = \{(\text{=}), (+\epsilon)\}$
<i>relop:\leq</i>	$S' = \{(-\epsilon), (\text{=})\}$	$S' = \{(+\epsilon)\}$

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, $S_N = \{\text{t}, \text{f}\}$. For each leaf node that is a relational expression, $S_N = \{(+\epsilon), (\text{=}), (-\epsilon)\}$.

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.

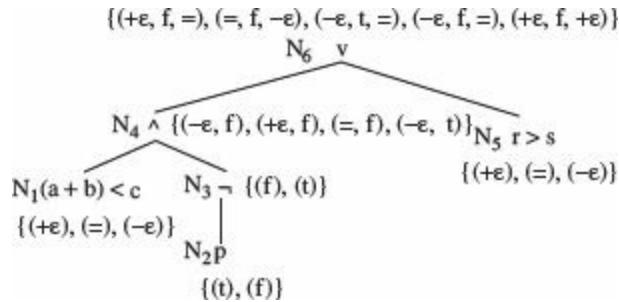


Figure 4.13 BRE constraint set for predicate $(a + b) < c \wedge \neg p \vee (r > s)$.

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	=	$< a = 1, b = 1, c = 1, p = \text{false}, r = 1, s = 1 >$
t_2	=	f	$- \epsilon$	$< a = 1, b = 0, c = 1, p = \text{false}, r = 1, s = 2 >$
t_3	$- \epsilon$	t	=	$< a = 1, b = 1, c = 3, p = \text{true}, r = 1, s = 1 >$
t_4	$- \epsilon$	f	=	$< a = 0, b = 2, c = 3, p = \text{false}, r = 0, s = 0 >$
t_5	$+ \epsilon$	f	$+ \epsilon$	$< a = 1, b = 1, c = 1, p = \text{false}, r = 2, s = 0 >$

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 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.

While constraint sets for non-singular predicates can be constructed directly by applying the procedures given earlier, more effective procedures are available.

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.

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.

Predicate (p_r)	DNF	Mutually singular components of p_r
$ab(b + c)$	$abb + abc$	$a; b(b + c)$
$a(bc + bd)$	$abc + abd$	$a; (bc + bd)$
$a(bc + \bar{b} + de)$	$abc + a\bar{b} + ade$	$a; bc + \bar{b}; de$

There are standard algorithms for converting expressions from DNF to CNF form and vice-versa. These algorithms are not described in this book.

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.

Procedure for generating a minimal constraint set from a predicate possibly containing non-singular expressions.

Input: A Boolean expression $E = e_1 + e_2 + \dots + e_n$ in minimal disjunctive normal form containing n terms. Term e_i , $1 \leq i \leq n$ 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 , $1 \leq i \leq n$, construct T_{e_i} as the set of constraints that make e_i true.

Step 2 Let $TS_{e_i} = T_{e_i} - \bigcup_{j=1, j \neq i}^n T_{e_j}$. Note that for $i \neq j$, $TS_{e_i} \cap TS_{e_j} = \emptyset$.

Step 3 Construct S_E^t by including one constraint from each TS_{e_i} ,

$1 \leq i \leq n$. Note that for each constraint $c \in S_E^t$, $E(c) = \text{true}$.

Step 4 Let e'_i denotes the term obtained by complementing the j^{th} literal in term e_i , for $1 \leq i \leq n$ and $1 \leq j \leq l_i$. We count the literals in a term from left to right, the leftmost literal being the first. Construct $F_{e'_i}$ as the set of constraints that make e'_i true.

- Step 5 Let $FS_{e_i} = F_{e_i} - \bigcup_{k=1}^n T_{e_k}$. Thus, for any constraint $c \in FS_{e_i}$, $E(c) = \text{false}$.
- Step 6 Construct S_E^f that is minimal and covers each FS_{e_i} 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

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 $(x \text{ OR } y \text{ or } z)$ 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.

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_{e1} and T_{e2} .

$$\begin{aligned} T_{e_1} &= \{(t, t, t, t), (t, t, t, f)\} \\ T_{e_2} &= \{(t, f, t, t), (t, f, f, t)\} \end{aligned}$$

Next we construct TS_{e1} and TS_{e2} .

$$\begin{aligned} TS_{e_1} &= \{(t, t, t, t), (t, t, t, f)\} \\ TS_{e_2} &= \{(t, f, t, t), (t, f, f, t)\} \end{aligned}$$

Notice that as T_{e1} and T_{e2} are disjoint, we get $TS_{e1} = T_{e1}$ and $TS_{e2} = T_{e2}$. Selecting one test each from TS_{e1} and TS_{e2} , we obtain a minimal set of constraints that make E true and cover each term of E as follows:

$$S_E^t = \{(t, t, t, f), (t, f, f, t)\}$$

Note that there exist four possible S_E^t . Next we construct constraints $F_{e_i^j}$, $1 \leq i \leq 2$, $1 \leq j \leq 3$. We have three terms corresponding to e_1 , namely

$e_1^1 = \bar{a}bc$, $e_1^2 = \bar{a}\bar{b}c$, and $e_1^3 = ab\bar{c}$. Similarly, we get three terms for e_2 , namely,

$e_2^1 = \bar{a}\bar{b}d$, $e_2^2 = abd$, and $e_2^3 = a\bar{b}\bar{d}$.

$$\begin{aligned} F_{e_1^1} &= \{(f, t, t, t), (f, t, t, f)\} \\ F_{e_1^2} &= \{(t, f, t, t), (t, f, t, f)\} \\ F_{e_1^3} &= \{(t, t, f, t), (t, t, f, f)\} \\ F_{e_2^1} &= \{(f, f, t, t), (f, f, f, t)\} \\ F_{e_2^2} &= \{(t, t, t, t), (t, t, f, t)\} \\ F_{e_2^3} &= \{(t, f, t, f), (t, f, f, f)\} \end{aligned}$$

Next we remove from the above six sets any test cases that belong to any of the sets T_{e_k} , $1 \leq k \leq n$ sets.

$$\begin{aligned}
FS_{e_1^1} &= F_{e_1^1} \\
FS_{e_1^2} &= \{(t, f, t, f)\} \\
FS_{e_1^3} &= F_{e_1^3} \\
FS_{e_2^1} &= F_{e_2^1} \\
FS_{e_2^2} &= \{(t, t, f, t)\} \\
FS_{e_2^3} &= F_{e_2^3}
\end{aligned}$$

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 .

$$S_E^f = \{(f, t, t, f), (t, f, t, f)(t, t, f, t), (f, f, t, t)\}$$

The set of constraints S_E generated by the MI-CSET procedure for expression E contains a total of six constraints as follows:

$$S_E = \{(t, t, t, f), (t, f, f, t), (f, t, t, f), (t, f, t, f), (t, t, f, t), (f, f, t, t)\}$$

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.

Procedure for generating a minimal constraint set for a predicate possibly containing non-singular expressions.

- | | |
|----------------|--|
| <i>Input:</i> | A Boolean expression E . |
| <i>Output:</i> | 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
 $E = \{E_1, E_2, \dots, E_n\}$.
- 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 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.

The following example illustrates the BOR-MI-CSET procedure.

Example 4.18 As in [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. 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 $e_2 = (bc + \bar{b}d)$ is a non-singular expression. In accordance with [Step 2](#), use the BOR-CSET procedure to find S_{e_1} as follows:

$$S_{e_1}^t = \{(t)\}; S_{e_1}^f = \{(\bar{f})\}$$

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 $v = \bar{b}\bar{d}$. Applying [Step 1](#) we obtain the following:

$$\begin{aligned} T_u &= \{(t, t, t), (t, t, \bar{f})\} \\ T_v &= \{(\bar{f}, t, t), (\bar{f}, \bar{f}, t)\} \end{aligned}$$

Applying [Step 2](#) from the MI-CSET procedure to T_u and T_v , and then [Step 3](#), we obtain the following sets:

$$\begin{aligned} TS_u &= T_u \\ TS_v &= T_v \\ S_{e_2}^t &= \{(t, t, \bar{f}), (\bar{f}, t, t)\} \end{aligned}$$

Notice that there are several alternatives for $S_{e_2}^t$ from which we need to

select one. Next, derive the false constraint set $S_{e_2}^f$ by applying [Steps 4](#), [5](#), and [6](#). The complemented subexpressions needed in these steps are $u^1 = \bar{b}c$, $u^2 = b\bar{c}$, $v^1 = bd$, and $v^2 = \bar{b}\bar{d}$.

$$\begin{aligned}
F_{u^1} &= \{(f, t, t), (f, t, f)\} \\
F_{u^2} &= \{(t, f, t), (t, f, f)\} \\
F_{v^1} &= \{(t, t, t), (t, f, t)\} \\
F_{v^2} &= \{(f, t, f), (f, f, f)\} \\
FS_{u^1} &= \{(f, t, f)\} \\
FS_{u^2} &= \{(t, f, t), (t, f, f)\} \\
FS_{v^1} &= \{(t, f, t)\} \\
FS_{v^2} &= \{(f, t, f), (f, f, f)\} \\
S'_{e_2} &= \{(f, t, f), (t, f, t)\}
\end{aligned}$$

Here is a summary of what we have derived so far by applying [Steps 1, 2, and 3](#) from procedure BOR-MI-CSET.

$S'_{e_1} = \{t\}$	From procedure BOR-CSET.
$S'_{e_1} = \{f\}$	From procedure BOR-CSET.
$S'_{e_2} = \{(t, t, f), (f, t, t)\}$	From procedure MI-CSET.
$S'_{e_2} = \{(f, t, f), (t, f, t)\}$	From procedure MI-CSET.

Now apply [Step 4](#) to the constraints sets generated for subexpressions e_1 and e_2 and obtain the BOR constraint set for the entire expression E . This process is illustrated in [Figure 4.14](#). The sets S'_E , S'_E , and S_E are listed below.

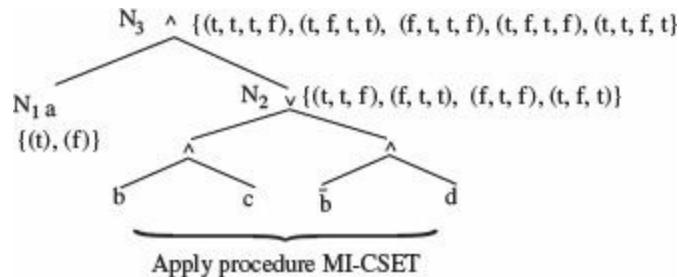


Figure 4.14 Constraint set for predicate $a(bc + \bar{b}d)$ derived using the BOR-MI-CSET procedure.

$$\begin{aligned}
S_E^t &= S_{e_1}^t \otimes S_{e_2}^t \\
&= \{(t, t, t, f), (t, f, t, t) \\
S_E^f &= (S_{e_1}^f \times \{t_2\}) \cup (\{t_1\} \times S_{e_2}^f) \\
&= \{(f, t, t, f), (t, f, t, f), (t, t, f, t)\} \\
S_E &= \{(t, t, t, f), (t, f, t, t), (f, t, t, f), (t, f, t, f), (t, t, f, t)\}
\end{aligned}$$

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 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.

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.

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.

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 . p_f has a Boolean operator fault created by the incorrect use of the \vee operator. Table 4.3 lists four test cases that form a BOR-adequate test set.

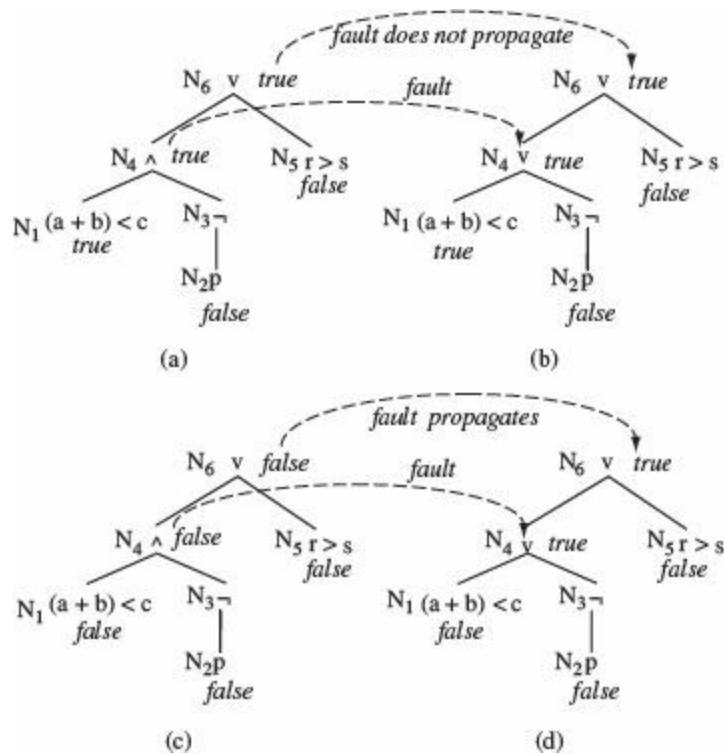


Figure 4.15 Abstract syntax trees in (a) and (b) show the failure of test t_1 to force the propagation of the \wedge operator fault. (c) and (d) show how test t_4 ensures the propagation of the \wedge operator fault.

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.

$$S = \{(f, f, f), (f, f, t), (f, t, f), (f, t, t), (t, f, f), (t, f, t), (t, t, f), (t, t, t)\}$$

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.

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	$Task_2$
.	
.	
.	
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.

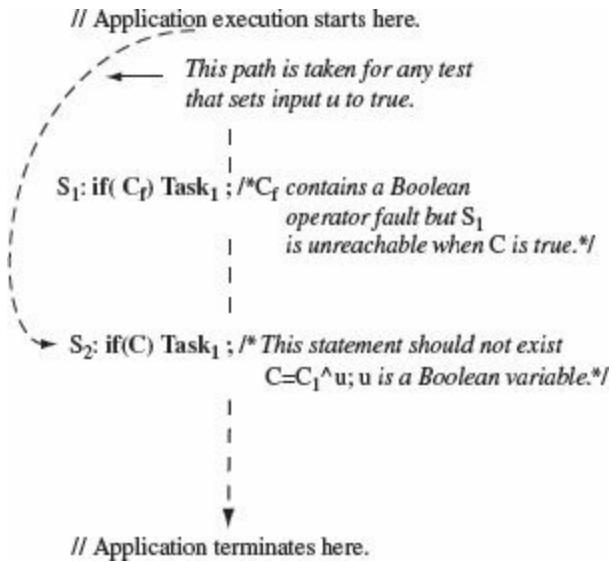


Figure 4.16 An implementation to illustrate why a predicate test that is theoretically guaranteed to detect any Boolean operator fault, might fail to do so in practice.

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 $Task_1$.

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 $u = \text{false}$ and the BOR constraint (t, t) , the error is revealed.

Test $a < b \wedge c > d$		Observed behavior	Expected behavior
true	(t, t)	Execute $Task_1$	Execute $Task_1$.
true	(t, f)	$Task_1$ not executed.	Do not execute $Task_1$.
true	(f, t)	$Task_1$ not executed.	Do not execute $Task_1$.
false	(t, t)	$Task_1$ not executed.	Execute $Task_1$.

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.

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

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.

Tests can be derived for predicates obtained from the requirements or from the code.

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

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.

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.

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.

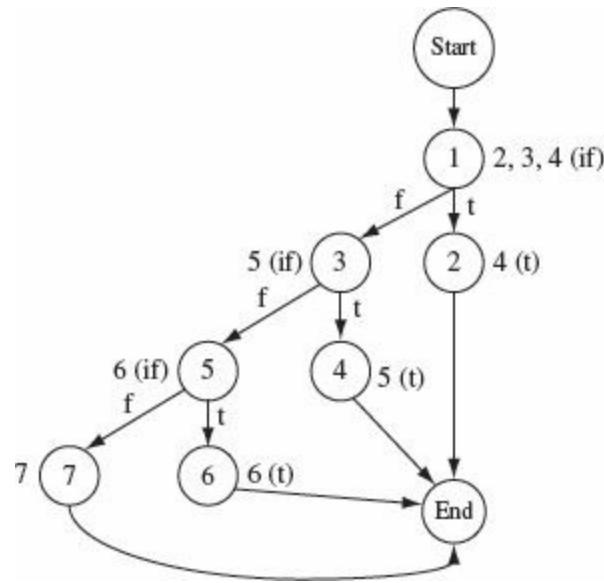


Figure 4.17 CFG for Program P4.1. Numbers against each node in the CFG correspond to the lines of code that belong to the corresponding basic block. (t) indicates the portion of the code executed when the condition at the indicated line is true, *t* and *f* label, respectively, the true and false branches.

- 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 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 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.

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, 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 and community creation. Once a community has been created, questions can be created and associated with this community.

Event/Activity	Response
<u>Start iDatify</u>	Login window visible.
<u>Login</u>	Home page for the user visible.
<u>Select “Create community”</u>	“New Community” form visible
<u>Fill the form, select “Create”</u>	Confirmation message displayed
<u>Select “Ask a question”</u>	“New Question” page visible
<u>Add a question with response fields.</u>	Create question visible
<u>Select “Submit”</u>	Question submitted confirmation message displayed.
<u>Select “My communities”</u>	Community just created is displayed.
<u>Select “Ask a New Question”</u>	“New Question” page visible
<u>Add another question with response fields.</u>	Create question visible
<u>Select “Submit”</u>	Question submitted confirmation message displayed.
<u>Select “My communities”</u>	Community just created is displayed.
<u>Select the community just created</u>	Community displayed. Both questions should be visible.

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](#).

```
1 String f(int x, int y){  
2     int q=0, z=0; String s= '' '';
```

```

3   if(y<x+1) // Error 1: this should be y<x+2
4     {s=s+‘‘1’’; q=y*2;}
5   if(q-x>0)
6     {s=s+‘‘2’’; z=z+1;}
7   if(z<=0) // Error 2: this should be z<0
8     s=s+‘‘3’’;
9   s=s+‘‘4’’;
10  return s;
11 }

```

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.

```

1 int f(double x, double y){
2   if(x*x<y) // Correct condition is x*x < y*y
3     return 1;
4   else
5     return 2;
6 }

```

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 ?

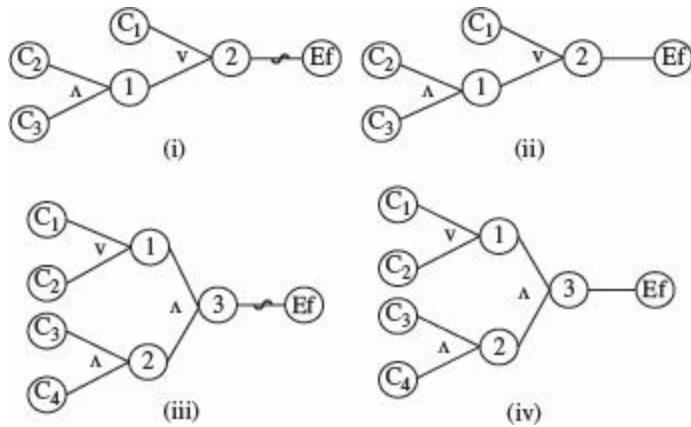


Figure 4.18 Cause-effect graphs for [Exercise 4.5](#).

4.6 Construct a sample implementation 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 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_{BOR} 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 be 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) $S_{e_i} \neq \emptyset$ for $1 \leq i \leq n$ and (b)

FS_{e_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 $E = a(bc + \bar{b}\bar{d})$ evaluate differently than E on at least one test case derived from S_E .

- (i) $a(bc + bd)$ Missing NOT operator.
- (ii) $a(\bar{b}c + \bar{b}\bar{d})$ Incorrect NOT operator.
- (iii) $a + (bc + \bar{b}d)$ Incorrect OR operator.
- (iv) $a(b\bar{c}\bar{b}d)$ Incorrect AND operator.
- (v) $a + (\bar{b}c + \bar{b}d)$ 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. $(a < b) \wedge (((r > s) \wedge u) \vee (a \geq b) \vee ((c < d) \vee (f = g)) \wedge (\bar{v} \wedge w));$ 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 operator.

- iv. $(a < b) \vee (((r > s) \wedge \bar{u}) \vee (a \geq b) \vee ((c < d) \vee (f = g)) \wedge (v \wedge w))$;
 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 .

- Derive a test set T_{CEG} to test the implementation of E using the test generation procedure described in [Section 4.3.3](#).

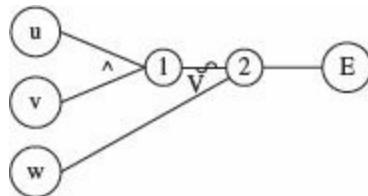


Figure 4.19 A cause-effect graph for [Exercise 4.25](#). Variables a, b, c and d are integers, $u : a < b$, $v : c > d$, and w is a Boolean variable.

- 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.
- 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.

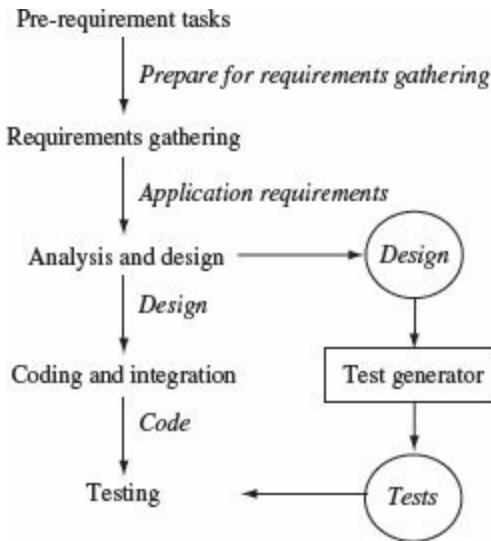


Figure 5.1 Design and test generation in a software development process. A *design* is an artifact generated in the Analysis and Design phase. This artifact becomes an input to the Test Generator algorithm that generates tests for input to the code during testing.

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 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 expressing 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 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.

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.

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\(b\)](#).

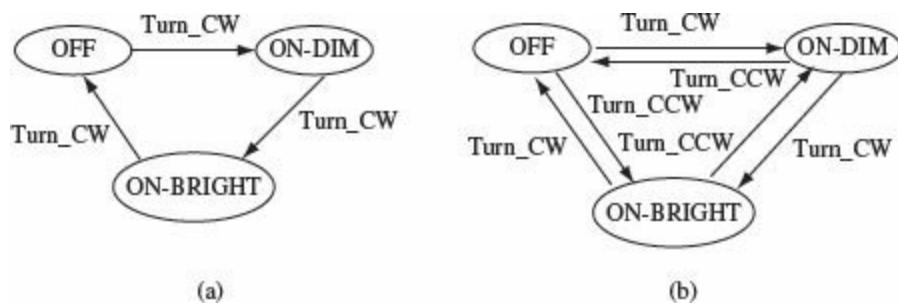


Figure 5.2 Change of lamp state in response to the movement of a switch. (a) Switch can only move one notch clockwise (CW). (b) Switch can move one notch clockwise and counterclockwise (CCW).

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 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 $10 * \text{num} + d$ 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.

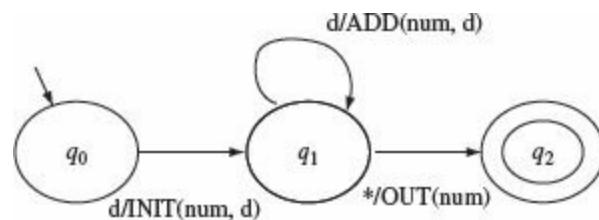


Figure 5.3 State diagram of the DIGDEC machine that converts sequence of decimal digits to an equivalent decimal number.

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.

FSMs are found in two types: Moore machines and Mealy machines. Moore machines do not associate actions with transitions while Mealy machines do.

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 usually abbreviated as NFSM or NDFSM or simply as NFA for Non-deterministic Finite Automata.

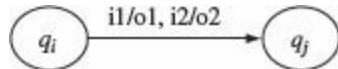


Figure 5.4 Multiple labels for an edge connecting two states in an FSM.

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 $s = a_1a_2 \dots a_n$ 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 $\delta(q_k, a_2a_3 \dots a_n) = q_j$. The output function is extended similarly. Thus, $O(q_i, s) = O(q_i, a_1) \cdot O(\delta(q_i, a_1), a_2, a_3, \dots, a_n)$. 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 $\delta(\text{OFF}, \text{Turn_CW}) = \text{ON_DIM}$. As an example of the output function, we get $O = \text{INIT}(\text{num}, 0)$ from [Figure 5.3](#).

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

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 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.

The transition function of an FSM can be described using a state diagram or a table.

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 i_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

$$r = \text{Turn_CCW Turn_CCW Turn_CW}$$

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.

$$\delta(\text{ON_BRIGHT}, \text{Turn_CCW}) = \text{ON_DIM}$$

$$\delta(\text{ON_DIM}, \text{Turn_CCW}) = \text{OFF}$$

$$\delta(\text{OFF}, \text{Turn_CW}) = \text{ON_DIM}$$

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 $\delta(\text{ON_BRIGHT}, r) = \text{ON_DIM}$ 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:

$$O(q_0, 1) = \text{INIT(num,1)}, \delta(q_0, 1) = q_1$$

$$O(q_1, 0) = \text{ADD(num,0)}, \delta(q_1, 0) = q_1$$

$$O(q_1, 0) = \text{ADD(num,0)}, \delta(q_1, 0) = q_1$$

$$O(q_1, 1) = \text{ADD(num,1)}, \delta(q_1, 1) = q_1$$

$$O(q_1, *) = \text{OUT(num)}, \delta(q_1, *) = q_2$$

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

$$\begin{aligned} O(q_0, 1001*) &= \text{INIT(num,1)} \text{ ADD(num,0)} \text{ ADD(num,0)} \text{ ADD(num,1)} \\ &\quad \text{OUT(num)} \\ \delta(q_0, 1001*) &= q_2 \end{aligned}$$

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				

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.

An FSM is complete, or completely specified, if there is exactly one transition in each state for every element in the input alphabet.

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 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()`.

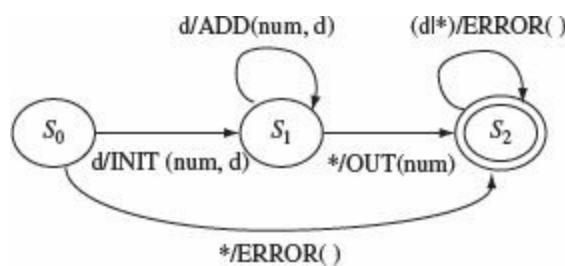


Figure 5.5 State diagram of a completely specified machine for converting a sequence of one or more decimal digits to their decimal number equivalent. $d|^*$ refers to digit or any other character.

5.3 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.

Testing an implementation against its design is also known as conformance testing.

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.

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.

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.

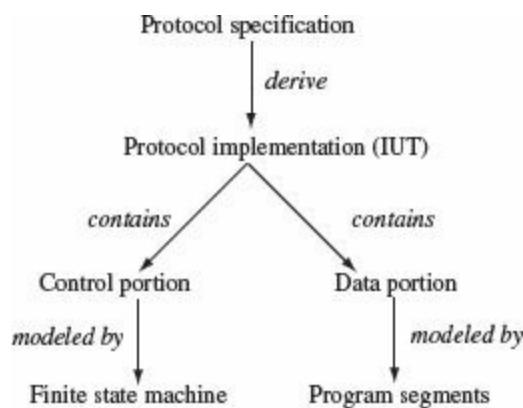


Figure 5.6 Relationship amongst protocol specification, implementation, and models.

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 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](#).

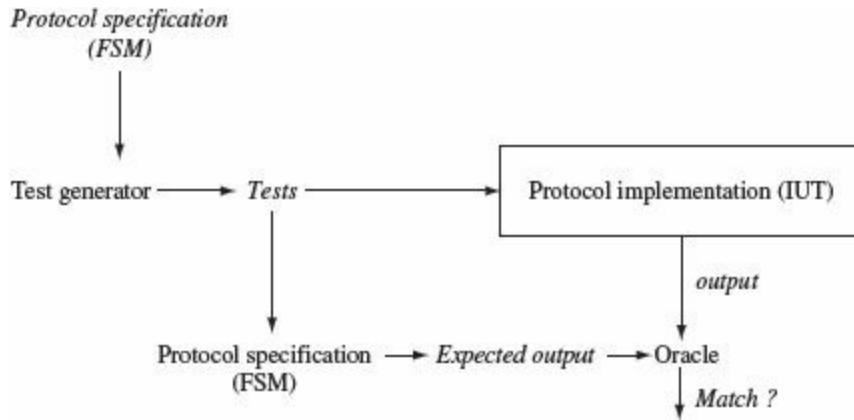


Figure 5.7 A simplified procedure for testing a protocol implementation against an FSM model. Italicized items represent data input to some procedure. Note that the model itself serves as data input to the test generator and as a procedure for deriving the expected output.

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.

A significant portion of this chapter is devoted to describing techniques for generating tests to test the control portion of an IUT corresponding 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

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 $T = \{t_1, t_2, \dots, t_n\}$, a test proceeds as follows:

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.

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:

$$X = X \cup \{Re\}$$

$$Y = Y \cup \{\text{null}\}$$

where Re denotes the reset input and null the corresponding output.

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.

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.

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](#).

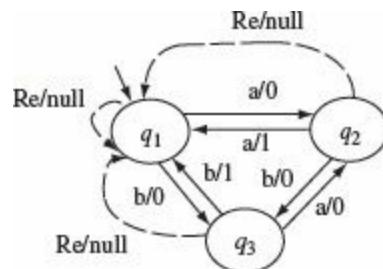


Figure 5.8 Transitions corresponding to reset (*Re*) inputs.

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.

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 .

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.

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.

- *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.

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.

- *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 , $\delta(q_i, s)$ 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.

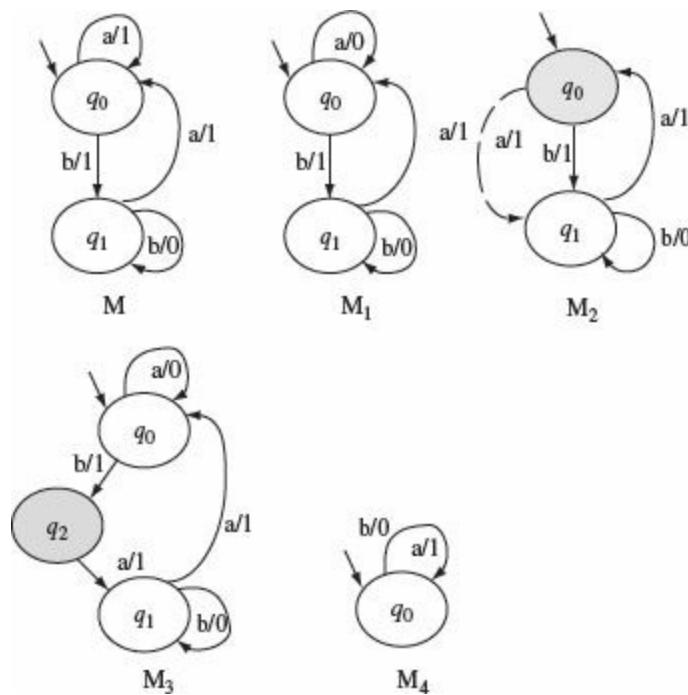


Figure 5.9 Machine M represents the correct design. Machines M_1 , M_2 , M_3 , and M_4 each contain an error.

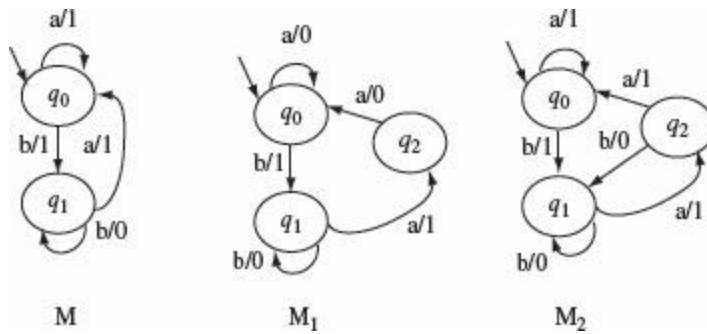


Figure 5.10 Machine M represents the correct design. Machines M₁ and M₂ each have an extra state. However, M₁ is erroneous and M₂ is equivalent to M.

A fault model aids in the assessment of the effectiveness of tests generated by techniques presented in this chapter.

- *Missing state error:* A missing state is another type of error. We note from Figure 5.9 that machine M₄ 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 $I(M_d)$ denote the set of all possible implementations of M_d . In order to make I finite, we assume that any implementation in $I(M_d)$ 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](#).

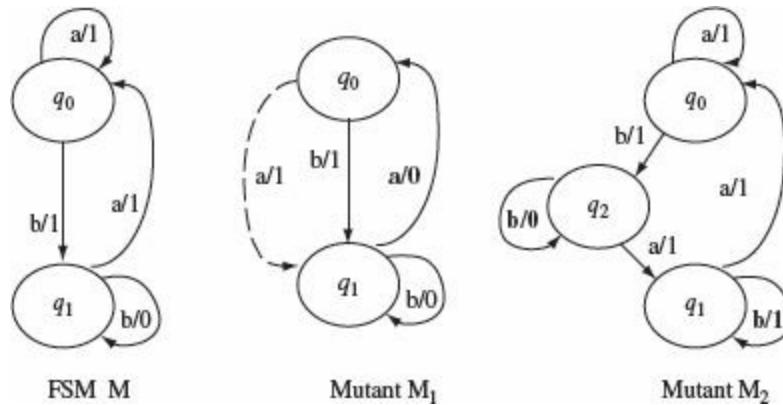


Figure 5.11 Two mutants of machine M . Mutant M_1 is obtained by introducing into M one operation error in state q_1 and one transfer error in state q_0 for input a . Mutant M_2 has been obtained by introducing an extra state q_2 in machine M and an operation error in state q_1 , on input b .

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.

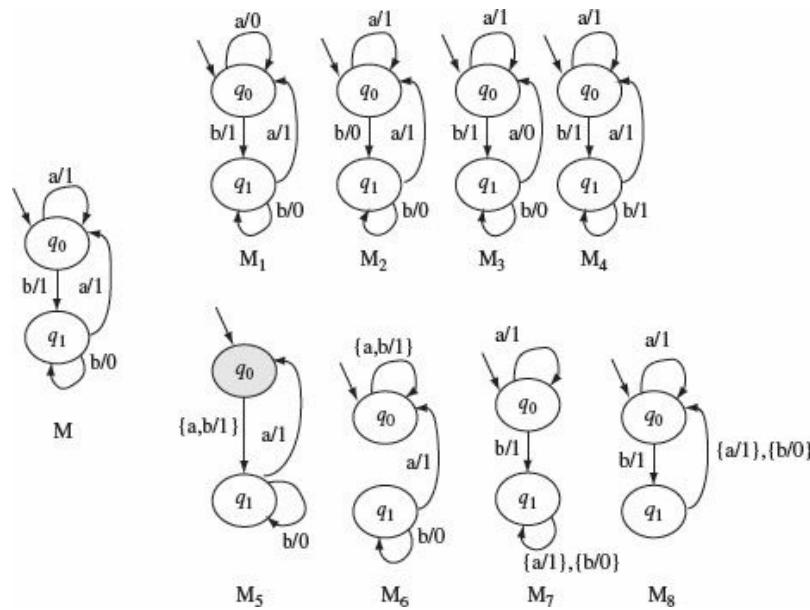


Figure 5.12 Eight first-order mutants of M generated by introducing operation and transfer errors. Mutants M_1 , M_2 , M_3 , and M_4 are generated by introducing operation errors in M . Mutants M_5 , M_6 , M_7 , and M_8 are generated by introducing transfer errors in M .

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 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 $|I(M)|$.

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 $I(M)$, is denoted by $FC(T, M)$ and computed as follows.

$$\begin{aligned} FC(T, M) &= \frac{\text{Number of mutants not distinguished by } T}{\text{Number of mutants that are not equivalent to } M} \\ &= \frac{N_t - N_e - N_f}{N_t - N_e} \end{aligned}$$

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

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.

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 .

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 $O(baaa, q_1) \neq O(baaa, q_2)$ 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.

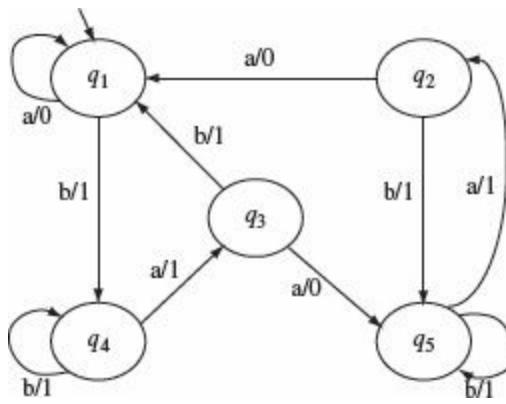


Figure 5.13 The transition and output functions of a simple FSM.

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, \dots, 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 $\Sigma_{k1}, \Sigma_{k2}, \dots, \Sigma_{kn}$ such that

- $\bigcup_{i=1}^n \Sigma_{ki} = Q$,
- States in Σ_{kj} , for $1 \leq j \leq n$, are *k-equivalent*,
- If $q_l \in \Sigma_{ki}$ and $q_m \in \Sigma_{kj}$, 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 = $\Sigma_{11} = \{q_1, q_2, q_3\}$

$$\text{Group 2} = \Sigma_{12} = \{q_4, q_5\}$$

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}
	q_4	q_{32}	q_{43}
3	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}
	q_4	q_{32}	q_{43}
3	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}
	q_2	q_{11}	q_{55}
2	q_3	q_{55}	q_{11}
	q_4	q_{33}	q_{44}
3	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 $P = \{P_1, P_2, \dots, P_n\}$ is the

- set of k -equivalence partition tables for $k = 1, 2, \dots, n$. 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 $z = x_0x_1 \dots x_r$ 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 $m = r, r - 1, r - 2, \dots, 1$.
 - 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 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 $O(q_1, \text{baaa}) = 1101$. 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.

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

5.5.3 Identification sets

Consider an FSM $M = (X, Y, Q, q_1, \delta, O)$ 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 .

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 $W_5 = \{a, aaa\}$.

The W - and the W_p -methods are both used for generating test sets from a minimal and complete FSM. The W_p -method will likely generate a smaller test set.

While the characterization sets are used in the W -method, the W_i sets are used in the W_p -method. The W - and the W_p -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](#).

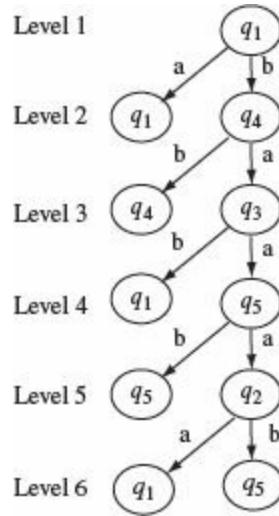


Figure 5.14 Testing tree for the machine with transition and output functions shown in [Figure 5.13](#).

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.

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.

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.

$$P = \{\epsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\}$$

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.

$$Z = (X^0 \cdot W) \cup (X \cdot W) \cup (X^2 \cdot W) \dots \cup (X^{m-1-n} \cdot W) \cup (X^{m-n} \cdot W)$$

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 " \cdot " denotes the string concatenation operation. For convenience, we shall use the shorthand notation $X[p]$ to denote the following set union

$$\{\epsilon\} \cup X^1 \cup X^2 \dots \cup X^{p-1} \cup X^p.$$

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.

$$\begin{aligned} T = P \cdot Z &= \{\epsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\} \cdot \{a, aa, \\ &\quad aaa, baaa\} \\ &= \{a, aa, aaa, baaa, \\ &\quad aa, aaa, aaaa, abaaa, \\ &\quad ba, baa, baaa, bbaaa, \\ &\quad bba, bbaa, bbaaa, bbbaaa, \\ &\quad baa, baaa, baaaa, babaaa, \\ &\quad baba, babaa, babaaa, babbaaa, \\ &\quad baaa, baaaa, baaaaa, baabaaa, \\ &\quad baaba, baabaa, baabaaa, baabbaaa, \\ &\quad baaaa, baaaaa, baaaaaa, baaabaaa \\ &\quad baaaba, baaabaa, baaabaaa, baaabbaaa \\ &\quad baaaaa, baaaaaa, baaaaaaa, baaaabaaa\} \end{aligned}$$

Assuming that the IUT has one extra state, i.e. $m = 6$, we obtain Z and the test set $P \cdot Z$ as follows.

$$\begin{aligned} Z = X^0 \cdot W \cup X^1 \cdot W &= \{a, aa, aaa, baaa, aa, aaa, aaaa, abaaa, \\ &\quad ba, baa, baaa, bbaaa\} \end{aligned}$$

$$\begin{aligned} T = P \cdot Z &= \{\epsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\} \cdot \\ &\quad \{a, aa, aaa, baaa, aa, aaa, aaaa, abaaa, ba, baa, baaa, bbaaa\} \end{aligned}$$

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 [\(c\)](#).

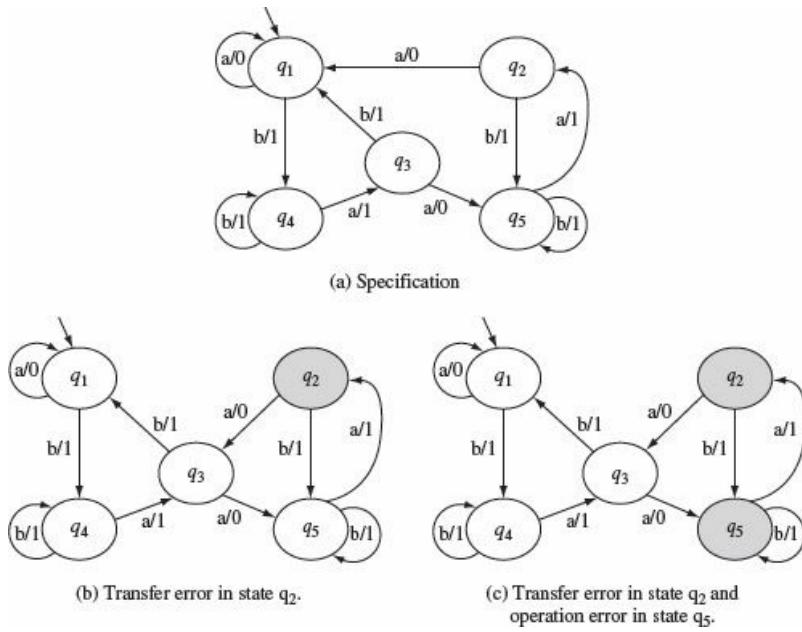


Figure 5.15 The transition and output functions of FSMs of the design under test, denoted as M in the text and copied from [Figure 5.13](#) and two incorrect designs in (b) and (c) denoted as M_1 , and M_2 in the text.

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 $baaaaaaa$ 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 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.

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.

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 .

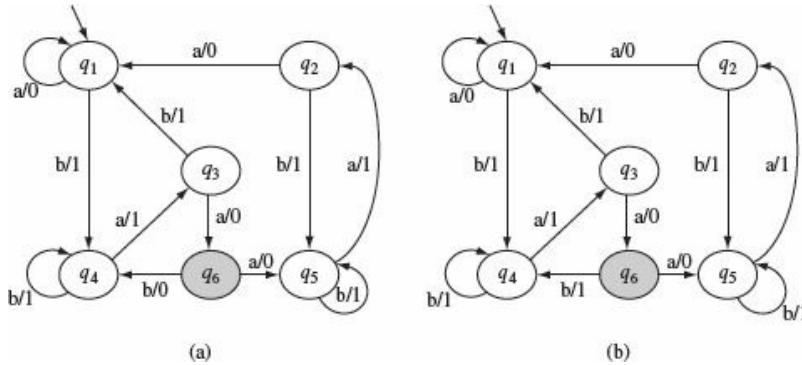


Figure 5.16 Two implementations each of the design in Figure 5.13 each having an extra state. Note that the output functions of the extra state q_6 are different in (a) and (b)

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

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.

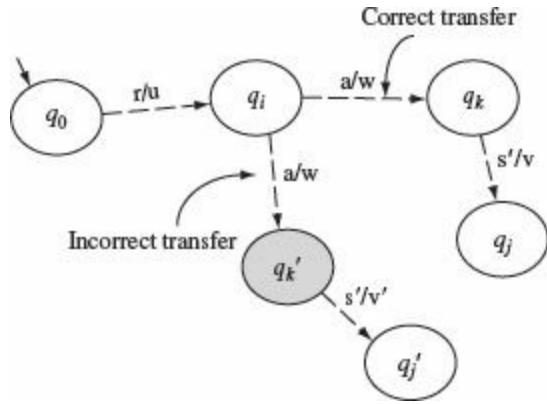


Figure 5.17 Detecting errors using tests generated by the W-method.

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.

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 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, $w_{v'} \neq w_v$. 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, 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](#).

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.

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 $q_t \in Q$, there exists an $r \in S$ such that $\delta(q_0, r) = q_t$. 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](#).

$$P = \{\epsilon, a, b, bb, ba, bob, baa, baab, baaa, baaab, baaaa\}$$

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 W be the set of all state identification sets of M, i.e.
$W = \{W_1, W_2, \dots, W_n\}$. |
| Step 4 | Let $R = \{r_{i_1}, r_{i_2}, \dots, r_{i_k}\}$ 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

$T_2 = R \otimes W = \bigcup_{j=1}^k (\{r_{i_j}\} \cdot W_{i_j})$, where $W_{i_j} \in W$ is the state identification set for state q_{i_j} .

End of Procedure for test generation using the Wp-method.

The \otimes 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](#).

$$T_1 = S \cdot W = \{a, aa, aaa, baaa, ba, baa, baaa, bbaaa, baa, baaa, baaaa, babaaa, baaa, baaaa, baaaaa, baabaaa, baaaa, baaaaa, baaaaaa, baaabaaa\}$$

Next, in accordance with [Step 4](#) we obtain R as

$$R = P - S = \{a, bb, bob, baab, baaab, baaaa\}.$$

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.

$$\begin{aligned}T_2 = R \otimes W &= (\{a\} \cdot W_1) \cup (\{bb\} \cdot W_4) \cup (\{bab\} \cdot W_5) \cup (\{baab\} \cdot W_5) \cup \\ &\quad (\{baaab\} \cdot W_5) \cup (\{baaaa\} \cdot W_1) \\ &= \{abaaa, aaa, aa\} \cup \{bba, bbaaa\} \cup \{baba, babaaa\} \cup \\ &\quad \{baaba, baabaaa\} \cup \{baaab, baaabaaa\} \cup \\ &\quad \{baaaabaaa, baaaaaa, baaaaa\} \\ &= \{abaaa, aaa, aa, bba, bbaaa, baba, babaaa, baaba, \\ &\quad baabaaa, baaaba, baaabaaa, baaaabaaa, \\ &\quad baaaaaa, baaaaa\}\end{aligned}$$

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 $R = P - S = \{r_{i_1}, r_{i_2}, \dots, r_{i_k}\}$. As before, $r_{ij} \in R$ moves M from its initial state to state q_{ij} , i.e. $\delta(q_0, r_{ij}) = q_{ij}$. Given R , we derive T_2 as follows.

$$T_2 = R \cdot X[m-n] \otimes W = \bigcup_{j=1}^k \{r_j\} \cdot (\bigcup_{u \in X[m-n]} u \cdot W_l),$$

where $\delta(q_0, r_{ij}) = q_{ij}$, $\delta(q_{ij}, u) = q_l$, and $W_l \in W$ 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 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.

$$\begin{aligned}
T_2 &= R \cdot X[m-n] \otimes \mathcal{W} \\
&= R \cdot X[1] \otimes \mathcal{W} \\
R \otimes \mathcal{W} &= (\{a\} \cdot W_1) \cup (\{bb\} \cdot W_4) \cup (\{baab\} \cdot W_5) \cup (\{bab\} \cdot W_1) \cup \\
&\quad (\{baaab\} \cdot W_1) \cup (\{baaaa\} \cdot W_5) \\
&= \{abaaa, aaa, aa, bba, bbaaa, babbbaa, babaa, baba, \\
&\quad baaba, baabaaa, baaaba, baaabaaa, baaaabaaa, \\
&\quad baaaaaa, baaaaa\} \\
(R \cdot X) \otimes \mathcal{W} &= \{aa, ab, bba, bbb, baba, babb, baaba, baabb, baaaba, \\
&\quad baaabb, baaaaa, baaaab\} \otimes \mathcal{W} \\
&= (aa \cdot W_1) \cup (ab \cdot W_4) \cup \\
&\quad (bba \cdot W_3) \cup (bbb \cdot W_4) \cup \\
&\quad (baaba \cdot W_2) \cup (baabb \cdot W_5) \\
&\quad (baba \cdot W_1) \cup (babb \cdot W_4) \\
&\quad (baaaba \cdot W_2) \cup (baaabb \cdot W_5) \\
&\quad (baaaaa \cdot W_1) \cup (baaaab \cdot W_5) \\
&= \{aabaaa, aaaa, aaa, aba, abaaa, bbaa, bbaaa, bbba, \\
&\quad bbbaaa, bababaaa, babaaa, babaa, babba, \\
&\quad babbaaa, baababaaa, baabaaa, baabaa, baabba, \\
&\quad baabbaaa, baaababaaa, baaabaaa, baaabaa, \\
&\quad baaabba, baaabbaaa, baaaaabaaa, baaaaaaa, \\
&\quad baaaaaa, baaaaba, baaaabaaa\}
\end{aligned}$$

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 \mathcal{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 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

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.

$$UIO(s) = i_1/o_1 \cdot i_2/o_2 \cdot \dots \cdot i_{(k-1)}/o_{(k-1)} \cdot i_k/o_k$$

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

$$in(UIO(s)) = i_1 \cdot i_2 \cdot \dots \cdot i_{(k-1)} \cdot i_k, \text{ and } out(UIO(s)) = o_1 \cdot o_2 \cdot \dots \cdot o_{(k-1)} \cdot o_k$$

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,

$$\delta(s, \text{in}(UIO(s))) = t, O(s, \text{in}(UIO(s))) = \text{out}(UIO(s)).$$

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.*

$$O(s, \text{in}(UIO(s))) \neq O(t, \text{in}(UIO(s))), \text{ for all } t \in Q, t \neq s.$$

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, 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](#).

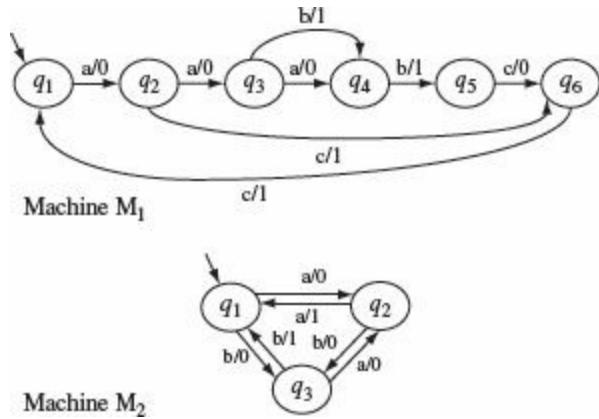


Figure 5.18 Two FSMs used in [Example 5.19](#). There is a UIO sequence for each state in Machine M₁. There is no UIO sequence for state q₁ in Machine M₂.

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

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 0₁ generated when $a \cdot c$ is applied to M1 in state q_1 . Similarly, applying ac to Machine M₁ 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 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 must be connected for the UIO sequence method to be applicable.

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.

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.

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	$b/1 \cdot a/1 \cdot b/1 \cdot b/1$
q_2	$a/0 \cdot b/1$
q_3	$b/1 \cdot b/1$
q_4	$a/1 \cdot b/1 \cdot b/1$
q_5	$a/1 \cdot a/0 \cdot b/1$

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 the UIO method will be discussed in [Section 5.8.9](#). In [Figure 5.19](#), the set of core edges is $\{(q_1, q_4), (q_4, q_1), (q_2, q_1), (q_2, q_5), (q_3, q_1), (q_3, q_5), (q_4, q_3), (q_5, q_2), (q_1, q_1), (q_2, q_2), (q_3, q_3), (q_4, q_4), (q_5, q_5)\}$.

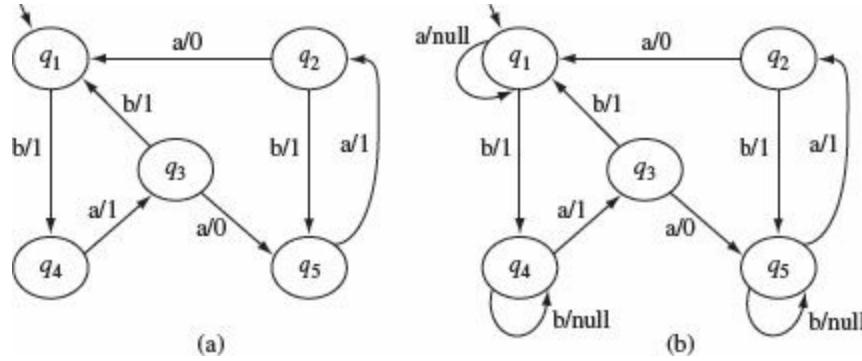


Figure 5.19 (a) An incomplete FSM. (b) Addition of null transitions to satisfy the completeness assumption.

The core-behavior of an FSM does not include self loops that generate null outputs.

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 $gen-1-uios$ to determine if $UIOs(s)$ consists of only one label.
- Step 5 If simple UIO sequence found then return $UIOs(s)$ and terminate this algorithm, otherwise proceed to the next step.
- Step 6 Apply the $gen-long-uios$ procedure to generate longer $UIOs(s)$.
- Step 7 If longer UIO sequence found then return $UIOs(s)$ and terminate this algorithm, else return with an empty $UIOs(s)$

End of procedure $gen-uios$.

Procedure $gen-1-uios$

- Input:* State $s \in Q$.
- Output:* $UIOs(s)$ 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 $UIOs(s) = label(e_i)$ otherwise return $UIOs(s) = ""$.

End of procedure $gen-1-uios$

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-*ui*o*
 indicating a failure to find any UIO for state *s*.

End of procedure gen-long-*ui*o

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-*ui*o (edge *te*)

Step 1 $k = k + 1$, $\text{Pattern}[k] = \text{Opattarn}[k] \cdot \text{label}(te)$. This could be a possible UIO.

Step 2 $\text{End}[k] = \text{tail}(te)$ and $\text{Led}[k] = \emptyset$.

Step 3 For each pair $oe \in \text{Oled}[i]$, where $h = \text{head}(oe)$ and $t = \text{tail}(oe)$, repeat the next step.

3.1 Execute the next step for each edge $o \in \text{Oedges}(t)$.

3.1.1 If $\text{label}(o) = \text{label}(te)$ then

$\text{Led}[k] = \text{Led}[k] \cup \{(head(oe), tail(o))\}$.

Step 4 If $\text{Led}[k] = \emptyset$, then UIO of length *L* has been found.
 Set $\text{UIO}(s) = \text{Pattern}[k]$ and terminate this and all procedures up until the main procedure for finding UIO for a state. If $\text{Led}[k] \neq \emptyset$, 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-*ui*o

5.8.5 Explanation of *gen-*ui*o*

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 $E(s) = e_1e_2 \dots e_k$ be a sequence of edges in M such that $s = \text{head}(e_1)$, $\text{tail}(e_i) = \text{head}(e_{i+1})$, and $1 \leq i < k$.

The gen-*uiio* algorithm uses the gen-1-*uiio* algorithm to generate a UIO of length 1. If such a UIO does not exist then it uses the gen-long-*uiio* 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.

For $E(s)$, we define a string of edge labels as $\text{label}(E(s)) = l_1 \cdot l_2 \cdot \dots \cdot l_{k-1} \cdot l_k$, where $l_i = \text{label}(e_i)$, $1 \leq i \leq k$, 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 $\text{label}(E(s)) \neq \text{label}(E(t))$, $s \neq t$ for all states t in M . If there is such an $E(s)$, then $\text{label}(E(s))$ is a UIO for state s . The uniqueness check is performed on $E(s)$ starting with $l = 1$ and continuing for $l = 2, 3, \dots$ 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-uiio* algorithm is explained with reference to [Figure 5.20](#). Algorithm *gen-uiio* for state s begins with the computation of some sets used in the subsequent steps. First, the algorithm computes $\text{Set}(el)$ for all distinct labels el in the FSM. $\text{Set}(el)$ is the set of all edges that have el as their label.

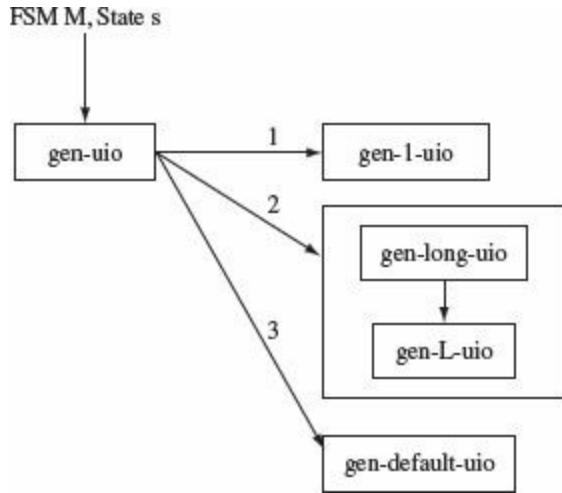


Figure 5.20 Flow of control across various procedures in gen-UIO.

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 each edge in $Oedges$ is saved in $Oend[e]$. The use of $Oled$ will become clear later in [Example 5.27](#).

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-ui* is called in an attempt to generate a UIO of length 1. For each edge $e_i \in Oedges$, *gen-1-ui* initializes $Opattern[i]$ to $label(e_i)$ and $Oend[i]$ to $tail(e_i)$. *Opattern* and *Oend* are used subsequently by *gen-ui* in case a UIO of length 1 is not found. *gen-1-ui* 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-ui*, *gen-ui* terminates if a UIO of length 1 was found, otherwise it invokes *gen-long-ui* in an attempt to find a UIO of length greater than 1. *gen-longer-ui* 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-ui* and *gen-1-ui*, 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-ui* concludes that state q_3 has no UIO of length 1 and returns to *gen-ui*.

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 .

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:

$$Tedges(q_2) = \{(2, 3), (2, 6)\}$$

The *gen-long-uio* procedure is invoked when there is no UIO sequence of length 1.

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 $Tedges$, $gen-L\text{-ui}o$ 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 $Oled[i]$, $gen-L\text{-ui}o$ 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 $gen\text{-ui}o$ procedure is terminated and $Pattern[te]$ is the desired UIO. If $Led[k]$ is not empty, then $gen-L\text{-ui}o$ returns to the caller for further attempts at finding $UIO(s)$.

Example 5.27 Continuing [Example 5.26](#), suppose that $gen-L\text{-ui}o$ is invoked with $i = 1$ and $te = (2, 3)$. The goal of $gen-L\text{-ui}o$ 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 $OutEdges(q_3)$ 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 is 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 $\text{gen-UIO}(q_1)$ 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 $\text{Oedges}(q_1)$.

Distinct labels in $M_2 = \{a/0, a/1, b/0, b/1\}$

$$\text{Set}(a/0) = \{(1, 2), (3, 2)\}$$

$$\text{Set}(a/1) = \{(2, 1)\}$$

$$\text{Set}(b/0) = \{(1, 3), (2, 3)\}$$

$$\text{Set}(b/1) = \{(3, 1)\}$$

$$\text{Oedges}(q_1) = \{(1, 2), (1, 3)\}$$

$$NE = 2$$

Next, as directed in [Step 3](#), we compute Oled and Oend for each edge in $\text{Oedges}(q_1)$. The edges in Oedges are numbered sequentially so that edge $(1, 2)$ is numbered 1 and edge $(1, 3)$ is numbered 2.

$$\text{Oled}[1] = \{(3, 2)\}$$

$$\text{Oled}[2] = \{(2, 3)\}$$

$$\text{Oend}[1] = q_2$$

$$\text{Oend}[2] = q_3$$

$$\text{Opattern}[1] = a/0$$

$$\text{Opattern}[2] = b/0$$

$\text{gen-1-UIO}(q_1)$ is now invoked. As $\text{Oled}[1]$ and $\text{Oled}[2]$ are nonempty, 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 $te = (2, 3)$. $gen-L\text{-ui}o$ is invoked with te as input and the value of index i as 1. Inside $gen-L\text{-ui}o$, 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 $gen-long\text{-ui}o$. The next iteration over $Tedges(q_2)$ is initiated with $te = (2, 1)$. $gen-L\text{-ui}o$ 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 $Oedges$ 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-uios*.

In *gen-long-uios*, 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*.

$$Opattern[1] = Pattern[1] = a/0 \cdot b/0$$

$$Opattern[2] = Pattern[2] = a/0 \cdot a/1$$

$$Opattern[3] = Pattern[3] = b/0 \cdot a/0$$

$$Opattern[4] = Pattern[4] = b/0 \cdot b/1$$

$$Oend[1] = End[1] = 3$$

$$Oend[2] = End[2] = 1$$

$$Oend[3] = End[3] = 2$$

$$Oend[4] = End[4] = 1$$

$$Oled[1] = Led[1] = (3, 3)$$

$$Oled[2] = Led[2] = (3, 1)$$

$$Oled[3] = Led[3] = (2, 2)$$

$$OIed[4] = Led[4] = (2, 1)$$

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.

$$\text{Set}(a/0) = \{(1, 2), (2, 3), (3, 4)\}$$

$$\text{Set}(b/1) = \{(3, 4), (4, 5)\}$$

$$\text{Set}(c/0) = \{(5, 6)\}$$

$$\text{Set}(c/1) = \{(2, 6), (6, 1)\}$$

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 $Oend$.

$$Oled[1] = \{(1, 2), (2, 3), (3, 4)_{a/0}\} - \{(1, 2)\} = \{(2, 3), (3, 4)_{a/0}\}.$$

$$Opattern[1] = label((1, 2)) = a/0, \text{ and}$$

$$Oend[1] = tail((1, 2)) = 2.$$

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 $t = \text{tail}((1, 2)) = 2$. $Tedges(2) = \{(2, 3), (2, 6)\}$.
- 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 $k = 1$, $Pattern[1] = Opattern[1] \cdot \text{label}((2, 3)) = a/0 \cdot a/0$.
- 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 $\text{label}((3, 4)_{a/0}) = \text{label}((2, 3))$, set $Led[1] = \{(2, 4)\}$. Next select $o = (3, 4)_{b/1}$ and execute [Step 3.1.1](#).
- Step 3.1.1 As $\text{label}((2, 3)) \neq \text{label}((3, 4)_{b/1})$, 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 $\text{label}((4, 5)) \neq \text{label}((3, 4)_{a/0})$ 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. |
| <i>gen-long-UIO</i> | <i>Input</i> : State q_1 |
| Step 2.2.2 | Select another element of $Tedges$ and invoke $gen-L-UIO(e, te)$. For this iteration $te = (2, 6)$. |
| <i>gen-L-UIO</i> | <i>Input</i> : $te = (2, 6)$ |
| 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 $\text{label}((4, 5)) \neq \text{label}((2, 6))$ 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-uiو* 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-uiو*.

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.

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-uios* algorithm fails.

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 .

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).

$$P_1(q_5) = c/0 \cdot c/1$$

$$P_5(q_2) = a/0 \cdot a/0 \cdot b/1 \text{ or } P_5(2) = a/0 \cdot b/1 \cdot b/1$$

$$P_6(q_1) = a/0 \cdot c/1$$

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- ui o* 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.

$$\begin{aligned} \text{Sig}(q_i) &= W(q_1, q_2) \cdot (P_1(t_2) \cdot W(q_1, q_3)) \cdot (P_1(t_3) \cdot W(q_1, q_4)) \dots \\ &\quad (P_1(t_{n-1}) \cdot W(q_1, q_n)), \quad \text{for } i=1. \\ &= W(q_i, q_1) \cdot (P_i(t_1) \cdot W(q_i, q_2)) \cdot (P_i(t_2) \cdot W(q_i, q_3)) \dots \\ &\quad (P_i(t_{i-2}) \cdot W(q_i, q_{i-1})) \cdot (P_i(t_{i-1}) \cdot W(q_i, q_{i+1})) \dots \\ &\quad (P_i(t_{n-1}) \cdot W(q_i, q_n)), \quad \text{for } i \neq 1. \end{aligned}$$

Example 5.32 We have seen earlier that *gen-long-uios* 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

$$TE(e) = P_{\text{head}(e)}(1) \cdot \text{label}(e) \cdot \text{UIO}(\text{tail}(e)).$$

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

$$TA = X_{e \in E}((Re/\text{null}) \cdot TE(e)).$$

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	$a/0 \cdot c/1$	null
q_2	$c/1 \cdot c/1$	$a/0$
q_3	$b/1 \cdot b/1$	$a/0 \cdot a/0$
q_4	$b/1 \cdot c/0$	$a/0 \cdot a/0 \cdot a/0$
q_5	$c/0$	$a/0 \cdot a/0 \cdot a/0 \cdot b/1$
q_6	$c/1 \cdot a/0$	$a/0 \cdot c/1$

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	$a/0 \cdot c/1 \cdot c/1$
2	q_1, q_1	$Re/null \cdot Re/null \cdot a/0 \cdot c/1$
3	q_2, q_3	$a/0 \cdot a/0 \cdot b/1 \cdot b/1$
4	q_2, q_6	$a/0 \cdot c/1 \cdot c/1 \cdot a/0$
5	q_2, q_1	$a/0 \cdot Re/null \cdot a/0 \cdot c/1$
6	q_3, q_1	$a/0 \cdot a/0 \cdot a/0 \cdot b/1 \cdot c/0$
7	q_3, q_4	$a/0 \cdot a/0 \cdot b/1 \cdot b/1 \cdot c/0$
8	q_3, q_4	$a/0 \cdot a/0 \cdot Re/null \cdot a/0 \cdot c/1$
9	q_3, q_1	$a/0 \cdot a/0 \cdot a/0 \cdot c/0$
10	q_4, q_5	$a/0 \cdot a/0 \cdot a/0 \cdot Re/null \cdot a/0 \cdot c/1$
11	q_4, q_1	$a/0 \cdot a/0 \cdot b/1 \cdot c/0 \cdot c/1$
12	q_5, q_6	$a/0 \cdot a/0 \cdot a/0 \cdot b/1 \cdot Re/null \cdot a/0 \cdot c/1$
13	q_5, q_1	$a/0 \cdot c/1 \cdot c/1 \cdot a/0 \cdot c/1$
14	q_6, q_1	$a/0 \cdot c/1 \cdot Re/null \cdot a/0 \cdot c/1$

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 $TE(e)$.

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](#).)

Tests generated using the UIO method can be reduced using a simple optimization Procedure.

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	$(q_1, q_1)_b/\text{null}$	$b/\text{null} \cdot a/0 \cdot c/1$
2	$(q_1, q_1)_c/\text{null}$	$c/\text{null} \cdot a/0 \cdot c/1$
3	$(q_2, q_2)_b/\text{null}$	$a/0 \cdot b/\text{null} \cdot c/1 \cdot c/1$
4	$(q_3, q_3)_c/\text{null}$	$a/0 \cdot a/0 \cdot c/\text{null} \cdot b/1 \cdot b/1$
5	$(q_4, q_4)_a/\text{null}$	$a/0 \cdot a/0 \cdot a/0 \cdot a/\text{null} \cdot b/1 \cdot c/0$
6	$(q_4, q_4)_c/\text{null}$	$a/0 \cdot a/0 \cdot a/0 \cdot c/\text{null} \cdot b/1 \cdot c/0$
7	$(q_5, q_5)_a/\text{null}$	$a/0 \cdot a/0 \cdot a/0 \cdot b/1 \cdot a/\text{null} \cdot c/0$
8	$(q_5, q_5)_b/\text{null}$	$a/0 \cdot a/0 \cdot a/0 \cdot b/1 \cdot b/\text{null} \cdot c/0$
9	$(q_6, q_6)_a/\text{null}$	$a/0 \cdot c/1 \cdot a/\text{null} \cdot c/1 \cdot a/0$
10	$(q_6, q_6)_b/\text{null}$	$a/0 \cdot c/1 \cdot b/\text{null} \cdot c/1 \cdot a/0$

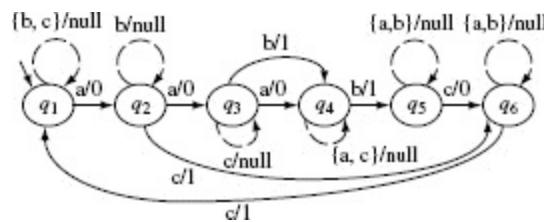


Figure 5.21 Machine M_1 , from [Figure 5.18](#) with all non-core edges shown as dashed lines.

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 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.

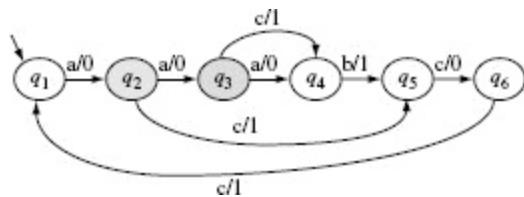


Figure 5.22 State diagram of an IUT containing two faults. State q_2 has a transfer error and q_3 has an operation error. The behavior of this IUT is to be tested against that of the machine in [Figure 5.21](#). Non-core edges are not shown.

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 $O(q_1, acca) = 010null$.

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 $O(q_1, accac) = 010null11$ 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 $O(q_1, aabbc) = 00nullnull11$ 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 (also a/0 · a/0)	b/1

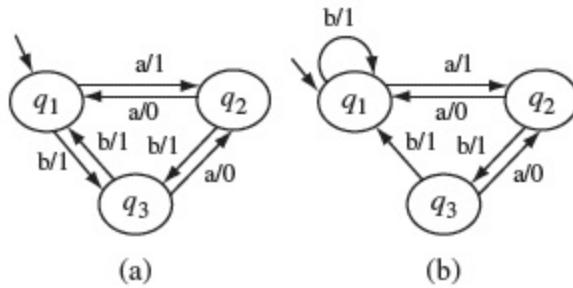


Figure 5.23 State diagram of an FSM for which a test sequence derived using the UIO approach does not reveal the error. (a) Specification FSM. (b) State diagram of a faulty IUT.

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	q1, q2	Re/null· a/1· a/0· a/1
5	q1, q3	Re/null· b/1· b/1· a/1
6	q2, q1	Re/null· a/1· a/0· a/1
7	q2, q3	Re/null· a/1· b/1· b/1· a/1
8	q3, q1	Re/null· b/1· b/1· a/1
9	q3, q2	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.

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.

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.

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.

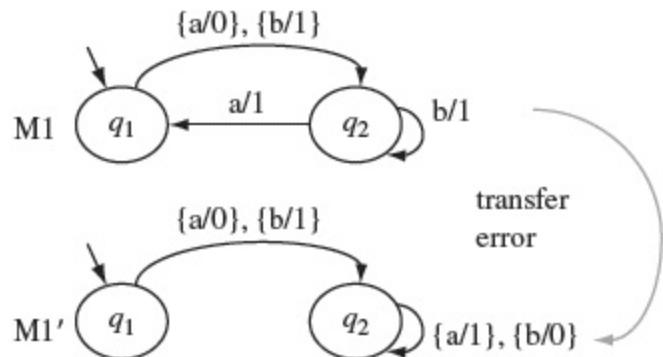


Figure 5.24 Machine M'_1 contains a transfer fault with respect to machine M_1 in state q_2 .

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 $\delta_{M_1}(q_l, t) = 0111$ and $\delta_{M'_1}(q_l, t) = 0100$. 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.

$$S = \{(tr_1, tr_2), (tr_1, tr_3), (tr_2, tr_2), (tr_2, tr_3), (tr_3, tr_4), (tr_3, tr_5), \\ (tr_4, tr_4), (tr_4, tr_5), (tr_5, tr_6), (tr_5, tr_1), (tr_6, tr_4), (tr_6, tr_5)\}$$

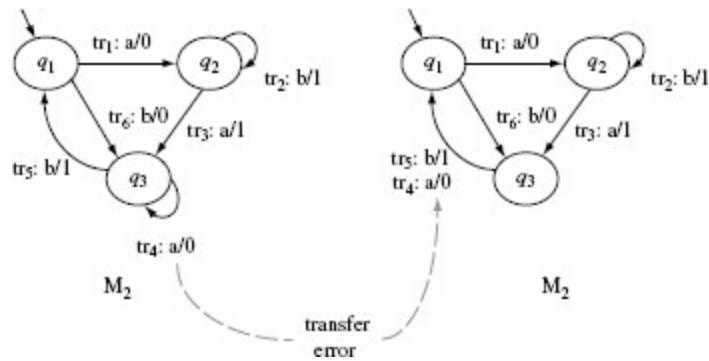


Figure 5.25 Machine M'_2 contains a transfer fault with respect to machine M_2 in state q_3 .

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.

$$tr_1 : \{(tr_1, tr_2), (tr_1, tr_3)\}$$

$$tr_2 : \{(tr_2, tr_1), (tr_2, tr_3)\}$$

$$tr_3 : \{(tr_3, tr_4), (tr_3, tr_5)\}$$

$$tr_4 : \{(tr_4, tr_3), (tr_4, tr_5)\}$$

$$tr_5 : \{(tr_5, tr_6), (tr_5, tr_1)\}$$

$$tr_6 : \{(tr_6, tr_4), (tr_6, tr_5)\}$$

An *n*-switch 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 .

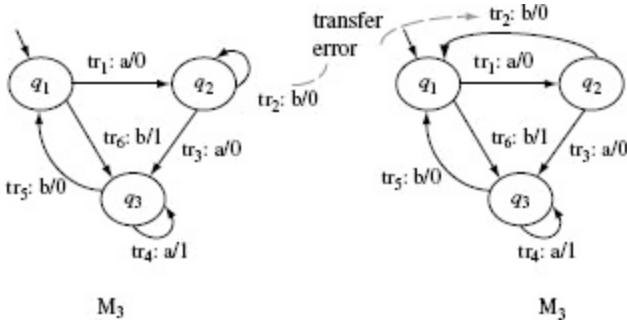


Figure 5.26 Machine M'_3 contains a transfer fault with respect to machine M_3 in state q_2 .

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 .

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.

$$T = \{aba, abb, aaa, aab, baa, bab, bba, bbb\}$$

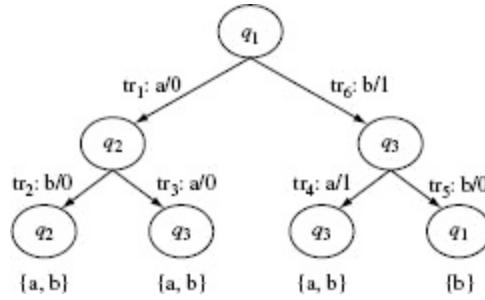


Figure 5.27 Testing tree of machine M_3 in [Figure 5.26](#).

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](#).

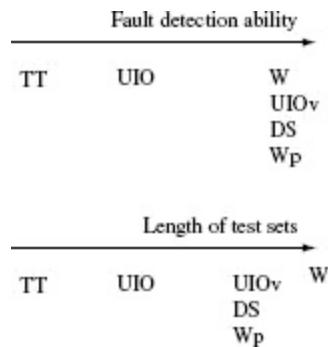


Figure 5.28 Relative fault detection effectiveness and length of the test sets of various automata theoretic techniques.

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.

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/>

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/>

Spec-Explorer, T-UPPAAL, ModelJUnit are publicly available tools for generating tests from timed automata, FSMs, and EFSMs.

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, \dots, 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 .

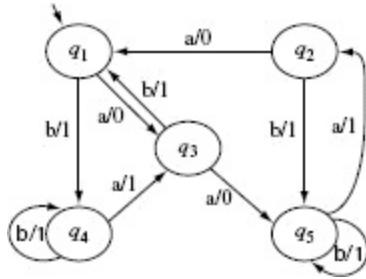


Figure 5.29 An implementation of M shown [Figure 5.15\(a\)](#) with a transfer error in state q_1 , on input a .

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?

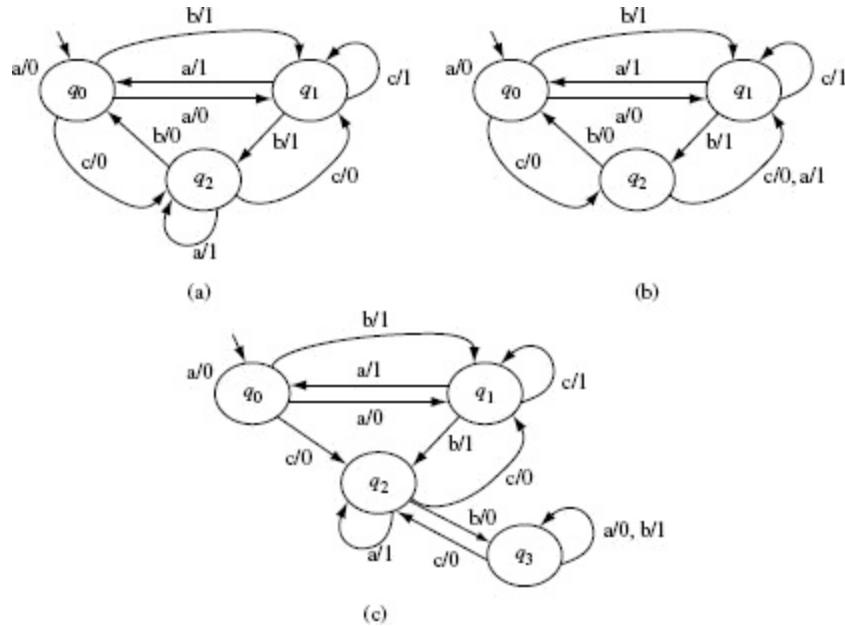


Figure 5.30 Three FSMs. (a) Design specification machine, (b) and (c) are machine indicating an erroneous implementation of the FSM in (a).

- 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-uios*?
- 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 ?

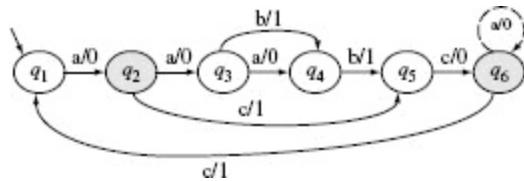


Figure 5.31 State diagram of an erroneous implementation of the machine in [Figure 5.21](#).

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 W_p-method, and show that each test set derived detects all errors shown in [Figure 5.32](#).

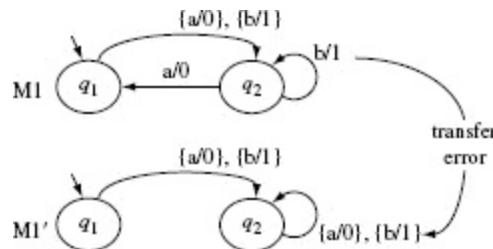


Figure 5.32 A transfer error.

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.)

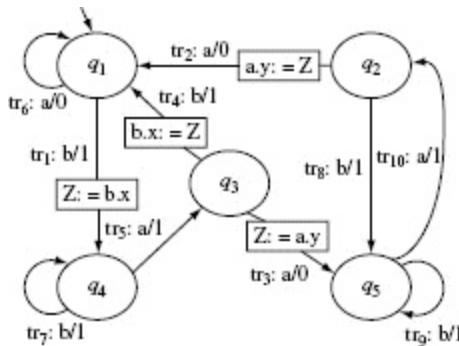


Figure 5.33 FSM of [Figure 5.13](#) augmented with definition and use of local variable Z.

- 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 M₂ shown in [Figure 5.25](#) that is adequate with respect to the 1-switch cover criterion. Which test in T distinguishes M₂ 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, \dots, 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 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.

Table 6.1 Factors and levels for the Unix sort utility.

Factor	Meaning	Levels			
		Unused	Used		
-	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 <i>output</i>	Output sent to a file.	Unused	Valid file	Invalid file	
-T <i>directory</i>	Temporary directory for sorting.	Unused	Exists	Does not exist	
-y <i>kmem</i>	Use <i>kmem</i> kilobytes of memory for sorting.	Unused	Valid <i>kmem</i>	Invalid <i>kmem</i>	
-z <i>recsize</i>	Specifies record size to hold each line from the input file.	Unused	Zero size	Large size	
-dfi <i>Mnr</i>	Perform dictionary sort.	Unused	fi	Mnr	fiMnr

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			
		Unused	Used		
<code>-f</code>	Ignore case.	Unused	Used		
<code>-i</code>	Ignore non-ASCII characters.	Unused	Used		
<code>-M</code>	Fields are compared as months.	Unused	Used		
<code>-n</code>	Sort input numerically.	Unused	Used		
<code>-r</code>	Reverse the output order.	Unused	Used		
<code>-b</code>	Ignore leading blanks when using <code>+pos1</code> and <code>-pos2</code> .	Unused	Used		
<code>-tc</code>	Use character <i>c</i> as field separator.	Unused	<i>c</i> ₁	<i>c</i> ₁ <i>c</i> ₂	
<code>-kkeydef</code>	Restricted sort key definition.	Unused	start	end	starttype
<code>+pos1</code>	Start position in the input line for comparing fields.	Unused	f.c	f	0.c
<code>-pos2</code>	End position in the input line for comparing fields.	Unused	f.c	f	0.c
<i>file</i>	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 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.

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

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.

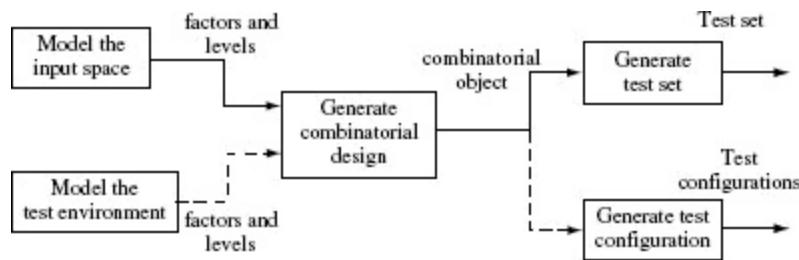


Figure 6.1 A process for generating tests and test configurations using combinatorial designs. A combinatorial design is represented as a combinatorial object which is an array of size $N \times k$ with N rows, one corresponding to at least one test run and k columns one corresponding to each factor.

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.

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](#).

```
< t1:File = Open, Edit = Paste, Typeset = MakeIndex >  
< t2:File = New, Edit = Cut, Typeset = 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 – oafile bfile>  
<  $t_2$  : sort – ocfile 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

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.

Tests generated using combinatorial designs aim at uncovering interaction faults. Such faults arise due to an interaction of two or more factors.

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 $t = 2$, 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.

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.

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

```
1 begin
2   int x, y, z;
3   input (x, y, z);
4   if(x==$x_1$ and y==$y_2$)
5     output (f(x, y, z));
6   else if(x==$x_2$ and y==$y_1$")
7     output (g(x, y));
8   else
9     output (f(x, y, z)+g(x, y)) // This statement
10  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, \dots, f_k , each at q_i , $1 \leq i \leq k$ levels, a vector V of factor levels is (l_1, l_2, \dots, 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*.

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.

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 don't care entry in a fault vector is indicated by an asterisk (*). Its 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 $(-1, -1, 0)$ 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 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 $t + 1, t + 2, \dots, t + k - 1$, and k -way runs. Hence, there is always a chance that runs generated with $t = 2$ reveal some higher level interaction faults.

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 .

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.

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.

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.

Example 6.11 Given $S = \{A, B\}$, here are two Latin squares of order 2.

$$\begin{array}{cc} A & B \\ B & A \end{array} \quad \begin{array}{cc} B & A \\ A & B \end{array}$$

Given $S = \{1, 2, 3\}$, we have the following three Latin Squares of order 3.

$$\begin{array}{ccc} 1 & 2 & 3 & 2 & 3 & 1 & 2 & 1 & 3 \\ 2 & 3 & 1 & 1 & 2 & 3 & 3 & 2 & 1 \\ 3 & 1 & 2 & 3 & 1 & 2 & 1 & 3 & 2 \end{array}$$

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.

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{array}$$

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.

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.

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{array}$$

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 based on integers $0, 1, \dots, n$ 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.

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.

6.5 Mutually Orthogonal Latin Squares

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 i^{th} 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 .

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.

Example 6.13 There are no mutually orthogonal Latin squares of order 2. Listed below are two mutually orthogonal Latin squares of order 3.

$$M_1 = \begin{matrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{matrix} \quad M_2 = \begin{matrix} 2 & 3 & 1 \\ 1 & 2 & 3 \\ 3 & 1 & 2 \end{matrix}$$

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 L 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 $\text{MOLS}(5) = \{M_1, M_2, M_3, M_4\}$. It is easy to check that indeed the elements of $\text{MOLS}(5)$ 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 method illustrated in the previous example is guaranteed to work only when constructing $\text{MOLS}(n)$ for n that is prime or a power of prime. For other values of n , the maximum size of $\text{MOLS}(n)$ is $n - 1$. However, different methods are used to construct $\text{MOLS}(n)$ 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 $\text{MOLS}(n)$ 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 .

The maximum number of $\text{MOLS}(n)$ is $n - 1$ when n is not prime.

6.6 Pairwise Design: Binary Factors

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.

A factor is considered binary if it can assume one of the two possible values.

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.

(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 number of tests needed to cover all binary factors is significantly less than what is obtained using exhaustive enumeration.

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, \dots, 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 $(X_1^*, X_2^*, \dots, X_n^*)$, 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 $(X_1^*, X_2^*, \dots, X_n^*)$, 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.

```

T = { t1: < Button = Create, Name = "", Symbol = "C",
      Atomic number = 6, Properties = "" >
      t2: < Button = Create, Name = "Carbon", Symbol = "C",
      Atomic number = -6, Properties = "Non-metal" >
      t3: < Button = Show, Name = "Hydrogen", Symbol = "C",
      Atomic number = 1, Properties = "Non-metal" >
      t4: < Button = Show, Name = "Carbon", Symbol = "C",
      Atomic number = 6, Properties = "" >
      t5: < Button = Show, Name = "", Symbol = "",
      Atomic number = -6, Properties = "Non-metal" >
      t6: < Button = Create, Name = "", Symbol = "",
      Atomic number = -6, Properties = "" >
}
  
```

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, \dots, F'_n denote the n factors.

$X^*_{i,j}$ denotes the j^{th} level of the i^{th} factor.

*/

Step 1 Relabel the factors as F_1, F_2, \dots, F_n such that the following ordering constraint is satisfied: $|F_1| \geq |F_2| \geq \dots \geq |F_{n-1}| \geq |F_n|$. Let $b=|F_1|$ and $k=|F_2|$. 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, \dots, 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, \dots, 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, \dots, 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

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.

Configurations generated using the algorithm PDMOLS may need to be revised when there are constraints among factors.

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.

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.

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	Netscape 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 $F_4 = F4'$, $b = k = 4$. Note that a different assignment is also possible because $|F_1| = |F_4|$ and $|F_2| = |F_3|$. 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, \dots, 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, \dots, 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.

Block	Row	$F_1(O)$	$F_2(B)$	$F_3(H)$	$F_4(P)$
1	1	1	1	1	1
	2	1	2	2	3*
	3	1	3	3*	4*
	4	1	4	4*	2
2	1	2	1	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	3	3	1	2
	4	3	4	2	4*
4	1	4	1	4*	4*
	2	4	2	3*	2
	3	4	3	2	1
	4	4	4	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.*

The test configurations constructed in [Step 5](#) do not satisfy the constraints among the factors. Hence, the table needs to be modified.

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 $(F_3 = 1, F_4 = 2)$.

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.

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.

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	<i>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.</i>
	2	1	2	1	1	
	3	1	3	1	2	
	4	1	4	2	2	
2	1	2	1	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	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	2	4	1	1	
5	1	–	2	2	1	
	2	–	1	2	2	
	3	3	3	–	2	
	4	4	4	–	1	

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 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.

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.

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.

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

Each row in an orthogonal array is also known as a run.

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 $MA(N, s_1^{k_1} s_2^{k_2}, \dots, s_p^{k_p}, t)$

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 $\sum_{i=1}^p k_i$.

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.

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.

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.

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.

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.

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 $\lambda = 1$.

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 $N \times t$ subarray contains each possible t -tuple at least once.

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.

$OA(8, 5, 2, 2) =$

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

Designs based on covering arrays are generally smaller than those based on orthogonal arrays.

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

$MCA(N, s_1^{k_1} s_2^{k_2}, \dots, s_p^{k_p}, t)$ and refers to an $N \times Q$ matrix of entries such that, $Q = \sum_{i=1}^p k_i$

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, $MA(12, 2^33^1, 2)$ covers all 20 of the 32 3-way interactions of the three binary and one ternary parameters. However, the mixed-level covering array $MCA(6, 2^33^1, 2)$ covers only 12 of the 20 3-way interactions. While **(Size=Large, Toppings= Custom, Address=Invalid)** is a 3-way interaction covered by $MCA(12, 2^33^1, 2)$, 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 designs.

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 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*.

The In-order-parameter (IPO) procedure is a practical means for constructing mixed-level covering arrays.

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, \dots, F_n denote n parameters. q_1, q_2, \dots, q_n denote the number of values for the corresponding parameter.

- T holds partial or complete runs of the form (v_1, v_2, \dots, v_k) , $1 \leq k \leq n$, where v_i denotes a value for parameter F_i , $1 \leq i \leq k$.
- $D(F)$ 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 $T = \{(r, s) | \text{for all } r \in D(F_1) \text{ and } s \in D(F_2)\}$.
- Step 2 [Possible termination] if $n=2$ then set $CA=T$ and terminate the algorithm, else continue.
- Step 3 [Add remaining parameters] Repeat the following steps for parameters F_k , $k=3, 4, \dots, n$.
- 3.1 [Horizontal growth] Replace each partial run $(v_1, v_2, v_{k-1}) \in T$ 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 $(v_1, v_2, \dots, v_j, \dots, v_{k-1}, v_k)$ to 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) T : A set of $m \geq 1$ runs of the kind
 $R = (v_1, v_2, \dots, v_{k-1})$, $k > 2$ 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: T' : A set of runs $(v_1, v_2, \dots, v_{k-1}, v_k)$, $k > 2$ obtained by extending the runs in T that cover the maximum number of pairs between parameter F_i , $1 \leq i \leq k-1$ and parameter F .

Procedure: HG

```
/*
 *  •  $D(F) = \{l_1, l_2, \dots, l_q\}$ ,  $q \geq 1$ .
 *  •  $t_1, t_2, \dots, t_m$  denote the  $m \geq 1$  runs in  $T$ .
 *  • For a run  $t \in T$ , where  $t = (v_1, v_2, \dots, v_{k-1})$ ,  $\text{extend}(t, v) = (v_1, v_2, \dots, v_{k-1}, v)$ , where  $v$  is a value of parameter  $F$ .
 *  • Given  $t = (v_1, v_2, \dots, v_{k-1})$  and  $v$  is a parameter value,
    $\text{pairs}(\text{extend}(t, v)) = \{(v_i, v), 1 \leq i \leq (k-1)\}$ .
*/
```

Step 1 Let $AP = \{(r, s) |$, where r is a value of parameter F_i , $1 \leq i \leq (k-1)$ and $s \in D(F)\}$. 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 $T' = \emptyset$. T' denotes the set of runs obtained by extending the runs in T in the following steps.

Step 3 Let $C = \min(q, m)$. Here C is the number of elements in the set T or the set $D(F)$, whichever is less.

Step 4 Repeat the next two sub steps for $j = 1$ to C .

4.1

Let $t'_j = \text{extend}(t_j, l_j)$. $T' = T' \cup t'_j$.

4.2

$$AP = AP - \text{pairs}(t'_j).$$

Step 5 If $C = m$, then return T' .

Step 6 We will now extend the remaining runs in 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 $v \in D(F)$.

6.2.1 $AP'' = \{(r, v) | r \text{ is a value in run } t_j\}$. 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 $t'_j = \text{extend}(t_j, v')$. $T' = T' \cup t'_j$.

6.4 $AP = AP - AP'$.

Step 7 Return T .

End of Procedure HG

Procedure for vertical growth.

Input: (a) T : A set of $m \geq 1$ runs each of the kind $(v_1, v_2, \dots, v_{k-1}, v_k)$, $k > 2$, 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 , $1 \leq i \leq (k-1)$, $s \in D(F_k)$, and the pair (r, s) is not contained in any run in T .

Output: A set of runs $(v_1, v_2, \dots, v_{k-1}, v_k)$, $k > 2$ 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

```
/*
 *  •  $\mathcal{D}(F) = \{l_1, l_2, \dots, l_q\}, q \geq 1$ .
 *  •  $t_1, t_2, \dots, t_m$  denote the  $m \geq 1$  runs in  $T$ .
 *  •  $(A_i.r, B_j.s)$  denotes a pair of values  $r$  and  $s$  that correspond to
parameters  $A$  and  $B$ , respectively.
 *  • In run  $(v_1, v_2, \dots, v_{i-1}, *, v_{i+1}, \dots, v_k), i < k$ , a “*” denotes a don’t
care value for parameter  $F_i$ . When needed, we use  $dc$  instead of
“*”.
*/
```

- Step 1 Let $T' = \emptyset$.
- 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 $(v_1, v_2, \dots, v_{i-1}, *, v_{i+1}, \dots, v_{k-1}, s) \in T'$, then replace it by the run $(v_1, v_2, \dots, v_{i-1}, r, v_{i+1}, \dots, s)$ and examine the next missing pair, else go to the next sub step.
 - 2.2 Let $t = (dc_1, dc_2, \dots, dc_{i-1}, r, dc_{i+1}, \dots, dc_{k-1}, s), 1 \leq i < k$.
Add t to T' .
- Step 3 For each run $t \in T'$, 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 $T \cup T'$.

End of Procedure VG

Example 6.23 Suppose it is desired to find a mixed covering design $MCA(N, 2^13^2, 2)$. 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:

$T = \{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2), (a_3, b_1), (a_3, b_2)\}$. We denote the elements of this set, starting from the first element, as t_1, t_2, \dots, 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.

$$AP = \{(a_1, c_1), (a_1, c_2), (a_1, c_3), (a_2, c_1), (a_2, c_2), (a_2, c_3), (a_3, c_1), (a_3, c_2), (a_3, c_3), (b_1, c_1), (b_1, c_2), (b_1, c_3), (b_2, c_1), (b_2, c_2), (b_2, c_3)\}$$

Next, in [HG.Step 2](#) initialize 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 T begins ith $j = 1$.

HG.Step 4.1: $t'_1 = \text{extend}(t_1, l_1) = (a_1, b_1, c_1)$. $T' = \{(a_1, b_1, c_1)\}$.

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: $t'_2 = \text{extend}(t_2, l_2) = (a_1, b_2, c_2)$. $T' = \{(a_1, b_1, c_1), (a_1, b_2, c_2)\}$.

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: $t'_3 = \text{extend}(t_3, l_3) = (a_2, b_1, c_3)$. $T' = \{(a_1, b_1, c_1), (a_1, b_2, c_2), (a_2, b_1, c_3)\}$.

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: $t'_4 = \text{extend}(t_4, c_1) = (a_2, b_2, c_1)$.

$$T' = \{(a_1, b_1, c_1), (a_1, b_2, c_2), (a_2, b_1, c_3), (a_2, b_2, c_1)\}.$$

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 A_P . 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: $t'_5 = \text{extend}(t_5, c_2) = (a_3, b_1, c_2)$.

$$T' = \{(a_1, b_1, c_1), (a_1, b_2, c_2), (a_2, b_1, c_3), (a_2, b_2, c_1), (a_3, b_1, c_2)\}.$$

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: $t'_6 = \text{extend}(t_6, c_3) = (a_3, b_2, c_3)$.

$$T' = \{(a_1, b_1, c_1), (a_1, b_2, c_2), (a_2, b_1, c_3), (a_2, b_2, c_1), (a_3, b_1, c_2), (a_3, b_2, c_3)\}.$$

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 T is now the same as the set 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 T .

VG.Step 1: We have $m = 6$, $F = C$, $D(F) = \{c_1, c_2, c_3\}$, $MP = U$, and $T' = \emptyset$. Recall that m is the number of runs in 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 T' is empty, hence move to the next step.

VG.Step 2.2: $t = (a_1, *, c_3)$. $T' = \{(a_1, *, c_3)\}$.

VG.Step 2: Next missing pair: $\{(A.a_2, C.c_2)\}$.

VG.Step 2.1: No run in T' is of the kind $(*, dc, c_2)$.

VG.Step 2.2: $t = (a_2, *, c_2)$. $T' = \{(a_1, *, c_3), (a_2, *, c_2)\}$.

VG.Step 2: Next missing pair: $\{(A.a_3, C.c_1)\}$.

VG.Step 2.1: No run in T' is of the kind $(*, dc, c_1)$.

VG.Step 2.2: $t = (a_3, *, c_1)$. $T' = \{(a_1, *, c_3), (a_2, *, c_2), (a_3, *, c_1)\}$. We are done with [Step 2](#).

VG.Step 3: Replace the don't cares in T' and to obtain the following.

$$T' = \{(a_1, b_2, c_3), (a_2, b_1, c_2), (a_3, b_1, c_1)\}$$

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 , $1 \leq i \leq 3$ and b_i by i , $1 \leq i \leq 2$. 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

Tool: AETG

Link: <http://aetgweb.argreenhouse.com/>

Description

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.

The Automatic Efficient Testcase Generator (AETG) was developed at Telcordia Technologies (now Applied Communications Sciences). It is 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.xhtml>

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 $S = \{\star, \circ, \bullet, \times, \Delta, \nabla\}$. (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, $D(A)$ and $D(B)$. This relationship can be generalized to the following.

If $X(F_1, F_2, \dots, F_n)$ then Y_1, Y_2, \dots, Y_m ; $n, m \geq 1$.

where X is a relation amongst one or more factors and Y s 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.

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?

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, \dots, 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.

Test adequacy refers to the “goodness” of a test set. The “goodness” ought to be measured against a quantitative criterion.

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 $T = \{t : \langle x = 2, y = 3 \rangle\}$ 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 $R_{2.2}$.

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 .

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.

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.

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.

A code-based coverage domain is derived through the analysis of the program under test.

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

```
1 begin
2     int x, y;
3     input (x, y);
4     sum=x+y;
5     output (sum);
6 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

```
1 begin
2     int x, y;
3     input (x, y);
4     if(x<y)
5         then
6             output (x+y);
7         else
8             output (x*y);
9 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

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.

Quantitatively measured inadequacy of a test set is an opportunity for its enhancement.

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 executed 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 test 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.

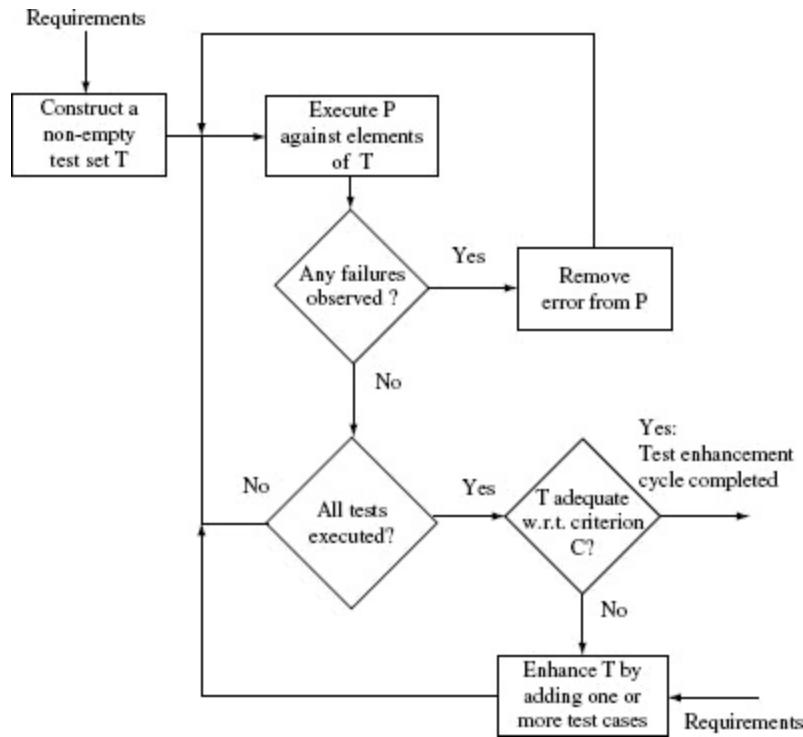


Figure 7.1 A test construction-enhancement cycle.

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

```

1  begin
2    int x, y;
3    int product, count;
4    input(x, y);
5    if (y≥0) {
6      product=1; count=y;
7      while (count>0) {
8        product=product*x;
9        count=count-1;
10     }// End of while
11   output(product);
12 } // End of true part of if
13 else
14   output ("Input does not match its
specification.");
15 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

$$T = \{\langle x = 0, y = 1 \rangle, \langle x = 1, y = 0 \rangle\}.$$

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.

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.

A black-box coverage criterion does not require an examination of the program under test in order to generate new tests.

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.

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](#).

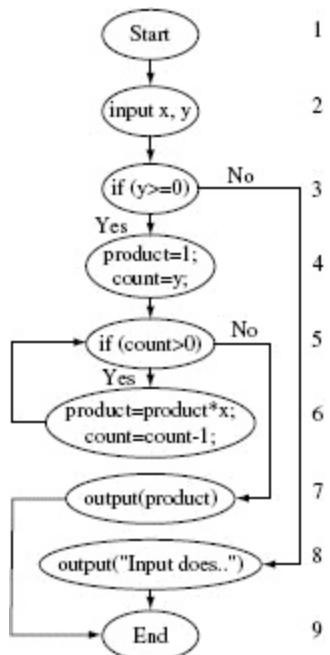


Figure 7.2 Control flow graph of P7.6.

p_1 : [1 → 2 → 3 → 4 → 5 → 7 → 9]; corresponds to $y \geq 0$ and loop body traversed zero times.

p_2 : [1 → 2 → 3 → 4 → 5 → 6 → 5 → 7 → 9]; corresponds to $y \geq 0$ and loop body traversed once.

p_3 : [1 → 2 → 3 → 8 → 9]; corresponds to $y < 0$ and the control reaches the output statement without entering the body of the while loop.

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.

The coverage domain for C' and P7.3 is $\{p_1, p_2, p_3\}$. Following the test enhancement procedure, we first measure the adequacy 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 $t : \langle x = 5, y = -1 \rangle$. 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

$$T = \{\langle x = 0, y = 1 \rangle, \langle x = 1, y = 0 \rangle, \langle x = 5, y = -1 \rangle\}.$$

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 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.

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.

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](#).

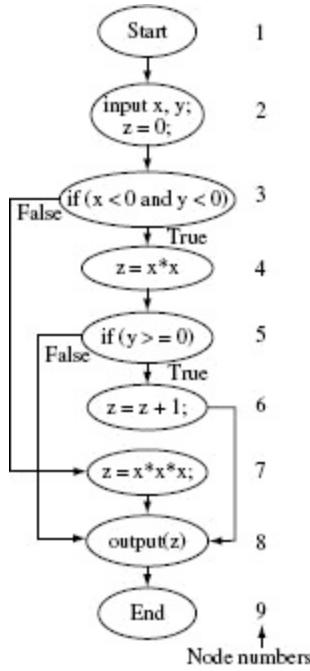


Figure 7.3 Control flow graph of P7.4.

p_1 : [1 → 2 → 3 → 4 → 5 → 6 → 8 → 9]; corresponds to conditions $x < 0$ and $y < 0$ and $y \geq 0$ evaluating to true.

p_2 : [1 → 2 → 3 → 4 → 5 → 8 → 9]; corresponds to conditions $x < 0$ and $y < 0$ evaluating to true and $y \geq 0$ to false.

p_3 : [1 → 2 → 3 → 7 → 8 → 9]; corresponds to $x < 0$ and $y < 0$ evaluating to false.

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.

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, 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*

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.

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.

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
8   (one of three possibilities).
9   product=1; // Initialize product.
10  // Set up the loop to accept and execute
11  requests.
12  while (true) {
13    // Process the “exponentiation” request.
14    if (request == exp){
15      input (x, y); count=y;
16      while (count > 0){
17        product=product*x; count=count-1;
18      }
19    } // End of processing the “exponentiation”
20    request.
21    // Process “factorial” request.
22    else if (request == fact){
23      input (x); count=x;
24      while (count >0){
25        product=product*count; count=count-1;
26      }
27    } // End of processing the “factorial” request.
28    // Process “exit” request.
29    elseif (request == exit){
30      output(“Thanks for using this program. Bye!”);
31      break; // Exit the loop.
32    } // End of if.
33    output(product); // Output the value of exponential
34    or factorial.
35    get_request (request); // Get user request once
36    again and jump to loop begin.
37  }
38 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.

```

 $T = \{ t_1 : <request = 1, x = 2, y = 3, request = 3>$ 
 $t_2 : <request = 2, x = 4, request = 3>$ 
 $t_3 : <request = 3>$ 
 $\}$ 

```

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.

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.

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 .

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.

$$T = \left\{ \begin{array}{lll} t_1 : & \langle \langle request = 1, x = 2, y = 3 \rangle \rangle \rightarrow & \\ & \langle request = 2, x = 4 \rangle \rangle & \rightarrow \langle request = 3 \rangle \rangle \end{array} \right\}$$

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:

$$T'' = \left\{ \begin{array}{lll} t_1 : & \langle \langle request = 1, x = 2, y = 3 \rangle \rangle \rightarrow & \langle request = 2, x = 4 \rangle \rangle \\ & \rightarrow \langle request = 3 \rangle \rangle \\ t_2 : & \langle \langle request = 2, x = 4 \rangle \rangle & \rightarrow \\ & \langle request = 1, x = 2, y = 3 \rangle \rangle & \rightarrow \langle request = 3 \rangle \rangle \end{array} \right\}$$

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.

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.

Programs that offer a GUI for user interaction are likely to be tested across multiple test cases in a single 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

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.

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).

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.

$$T_1 = \{t_1: \langle x = -1, y = -1 \rangle, t_2: \langle x = 1, y = 1 \rangle\}$$

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.

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.

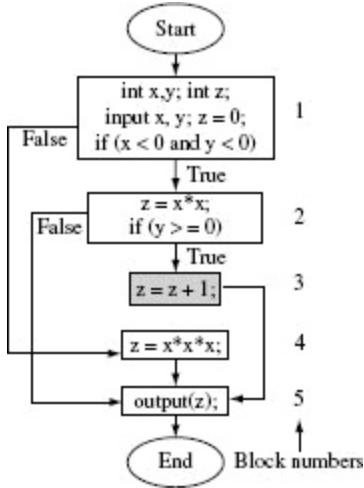


Figure 7.4 Control flow graph of 7.8. Blocks are numbered 1 through 5. The shaded block 3 is infeasible because the condition in block 2 will never be true.

$$T_2 = \left\{ \begin{array}{l} t_1: \langle x = -1 \quad y = -1 \rangle \\ t_2: \langle x = -3 \quad y = -1 \rangle \\ t_3: \langle x = -1 \quad y = -3 \rangle \end{array} \right\}$$

Some blocks in a coverage domain might be infeasible. Such infeasibility is usually discovered manually.

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.

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.

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.

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.

$if(A)$ $task 1 \text{ if } A \text{ is true;}$ $else$ $task 2 \text{ if } A \text{ is false;}$	$while(A)$ $task \text{ while } A \text{ is true;}$	$switch (e)$ $task 1 \text{ for } e=e1$ $else$ $task 2 \text{ for } e=e2$ \vdots $else$ $task n \text{ for } e=en$ $else$ $default \text{ task}$
(a)	(b)	(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.

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 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`.

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.

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, $C = (A \text{ and } B) \text{ or } (C \text{ and } A)$ 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 \text{ 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. A 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.

$$T = \{t_1 : \langle x = -5 \rangle\}$$

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' .

$$T' = \{t_1 : \langle x = -5 \rangle, t_2 : \langle x = 3 \rangle\}$$

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) ≠ foo-2 (3)` then the program will give an incorrect output when executed against test t_2 .

Decision coverage aids in testing if decisions in a program are correctly formulated and located.

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 $((x < 0 \text{ and } y < 0) \text{ or } (p \geq q))$. Logical operators and, or, and xor connect two or more simple conditions to form a *compound* condition. In addition, \neg (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, $(x < 0 \text{ and } y < 0)$ 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 $|C_c|/(|C_e| - |C_l|)$ 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:

$$\frac{|C_c|}{2 \times (|C_e| - |C_l|)}$$

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	<code>foo1(x, y)</code>
true	false	<code>foo2(x, y)</code>
false	true	<code>foo2(x, y)</code>
false	false	<code>foo1(x, y)</code>

Program P7.8

```

1 begin
2   int x, y, z;
3   input (x, y);
4   if(x<0 and y<0)
5     z=foo1(x,y);
6   else
7     z=foo2(x,y);
8   output(z);
9 end

```

Consider T designed to test [P7.8](#).

$$T = \{t_1 : \langle x = -3, y = -2 \rangle, t_2 : \langle x = -4, y = 2 \rangle\}$$

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 $t_3 : \langle x = 3, y = 4 \rangle$ to T . When executed against t_3 , P7.8 incorrectly computes z as $\text{foo2}(x, y)$. The output will be incorrect if $\text{foo1}(3, 4) \neq \text{foo2}(3, 4)$. 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

```
1 begin
2     int x, y, z;
3     input (x, y);
4     if(x<0 or y<0)
5         z=foo1(x,y);
6     else
7         z=foo2(x,y);
8     output(z);
9 end
```

$$T_1 = \left\{ \begin{array}{l} t_1: \langle x = -3, y = 2 \rangle \\ t_2: \langle x = 4, y = 2 \rangle \end{array} \right\}$$

$$T_2 = \left\{ \begin{array}{l} t_1: \langle x = -3, y = 2 \rangle \\ t_2: \langle x = 4, y = -2 \rangle \end{array} \right\}$$

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.

$$T = \left\{ \begin{array}{ll} t_1: & \langle x = -3 \quad y = -2 \rangle \\ t_2: & \langle x = 4 \quad y = 2 \rangle \end{array} \right\}$$

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 $D = (A < B) \text{ or } (A > C)$ 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](#).

Table 7.2 Combinations in $D = (A < B) \text{ or } (A > C)$.

	$A < B$	$A > C$	D
1	true	true	true
2	true	false	true
3	false	true	true
4	false	false	false

Now consider test set T containing two tests.

$$T = \left\{ \begin{array}{l} t_1: \langle A=2, B=3, C=1 \rangle \\ t_2: \langle A=2, B=1, C=3 \rangle \end{array} \right\}$$

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 .

$$T' = \left\{ \begin{array}{l} t_1: \langle A=2, B=3, C=1 \rangle \\ t_2: \langle A=2, B=1, C=3 \rangle \\ t_3: \langle A=2, B=3, C=5 \rangle \\ t_4: \langle A=2, B=1, C=1 \rangle \end{array} \right\}$$

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, \dots, 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 $(\sum_{i=1}^n 2^{k_i})$. 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 $|C_e| = \sum_{i=1}^n 2^{k_i}$ 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 f_1 through f_4 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.

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	$f4(A, B, C)$

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](#).

$$T = \left\{ \begin{array}{l} t_1: \langle A=2, B=3, C=1 \rangle \\ t_2: \langle A=2, B=1, C=3 \rangle \end{array} \right\}$$

[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](#).

$$T' = \left\{ \begin{array}{l} t_1: < A=2, B=3, C=1 > \\ t_2: < A=2, B=1, C=3 > \\ t_3: < A=2, B=3, C=5 > \end{array} \right\}$$

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 criterion.

[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 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'' .

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

$$T'' = \left\{ \begin{array}{l} t_1: < A=2, B=3, C=1 > \\ t_2: < A=2, B=1, C=3 > \\ t_3: < A=2, B=3, C=5 > \\ t_4: < A=2, B=1, C=1 > \end{array} \right\}$$

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

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.

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.

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.

$$T = \left\{ \begin{array}{l} t_1 : \langle x = -5, y = 2 \rangle \\ t_2 : \langle x = 9, y = 2 \rangle \end{array} \right\}$$

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 xy 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.

$$T = \left\{ \begin{array}{l} t_1: \langle x=5 \ y=0 \rangle \\ t_2: \langle x=5 \ y=2 \rangle \end{array} \right\}$$

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) \rightarrow (7, 10, 7) \rightarrow (7, 7, 11) \rightarrow (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.

$$T = \left\{ \begin{array}{l} t_1: \langle x=-5 \quad y=0 \rangle \\ t_2: \langle x=5 \quad y=2 \rangle \\ t_3: \langle x=-5 \quad y=2 \rangle \end{array} \right\}$$

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 $g(5) \geq 0$, the sequence of LCSAJs executed when P7.13 is executed against t_2 is $(1, 5, 7) \rightarrow (7, 10, \text{exit})$. 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.

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

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.

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 $C = (C_1 \text{ and } C_2) \text{ or } C_3$ 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](#).

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.

Table 7.6 Test cases for $C = (C_1 \text{ and } C_2) \text{ or } C_3$ to illustrate MC/DC coverage.
Identical rows are listed in parentheses.

Test	Inputs			Output
	C_1	C_2	C_3	
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.

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 MC-adequate tests for $C = (C_1 \text{ and } C_2) \text{ or } C_3$

Test	C_1	C_2	C_3	C	Effect demonstrated for
1[3]	true	false	true	true	C_3
2[4]	true	false	false	false	
3[11]	true	true	false	true	C_2
4[12]	true	false	false	false	
5[19]	true	true	false	true	C_1
6[20]	false	true	false	false	

[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](#).

Table 7.8 Minimal MC-adequate tests for $C = (C_1 \text{ and } C_2) \text{ or } C_3$

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	

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 $C = (C_1 \text{ and } C_2) \text{ or } C_3$ is tested. This is in contrast to only four tests required to satisfy the MC/DC criterion.

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 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.

Table 7.9 MC/DC adequate tests for compound conditions that contain two simple conditions.

Test	C_1	C_2	C	Comments
Condition: $C_a = (C_1 \text{ and } C_2)$				
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 .
<i>MC/DC adequate test set for $C_a = \{t_1, t_2, t_3\}$</i>				

Condition: $C_b = (C_1 \text{ or } C_2)$				
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 .
<i>MC/DC adequate test set for $C_b = \{t_4, t_5, t_6\}$</i>				

Condition: $C_c = (C_1 \text{ xor } C_2)$				
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 .
<i>MC/DC adequate test set for $C_c = \{t_7, t_8, t_9\}$</i>				

Test	C_1	C_2	C_3	C	Comments
t_1					
t_2					
t_3					
t_4					

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. $C = (C_1 \text{ and } C_2 \text{ and } C_3)$.

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.

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: $C_a = (C_1 \text{ and } C_2 \text{ and } C_3)$					
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: $C_b = (C_1 \text{ or } C_2 \text{ or } C_3)$					
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: $C_c = (C_1 \text{ xor } C_2 \text{ xor } C_3)$					
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 $C_c = \{t_9, t_{10}, t_{11}, t_{12}\}$					

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.
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 $C = (C_1 \text{ and } C_2 \text{ and } C_3)$ derived from tests for $C = (C_1 \text{ and } C_2)$.

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

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.

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.

1. Each block in P has been covered.
2. Each simple condition in P has taken both true and false values.
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

$$A = (p < q) \text{ or } (x > y)$$

are also included in the set of conditions to be covered. With regard to the fourth requirement, a condition such as $(A \text{ and } B) \text{ or } (C \text{ and } A)$ poses 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.

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

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, \dots, 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.

$$\text{MC}_c = \frac{\sum_{i=1}^N e_i}{\sum_{i=1}^N (n_i - f_i)}$$

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.

R₁: 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 $(\text{prev} = \text{"East"})$, where *prev* and *current* denote, respectively, the previous and current values of d .

R_{1.2}: Invoke fire-2 when $(x < y)$ and $(z * z \leq y)$ or $(\text{current} = \text{"South"})$.

R_{1.3}: Invoke fire-3 when none of the two conditions above is true.

R₂: The invocation described above must continue until an input Boolean variable becomes true.

Program P7.14

```

1 begin
2   float x,y,z;
3   direction d;
4   string prev,current;
5   bool done;
6   input(done);
7   current="North";
8   while (~done) {   ← Condition C1.
9     input(d);
10    prev=current; current=f(d);
11    input(x,y,z);
12    if((x<y) and (z * z > y) and (prev=="East"))
13      ← Condition C2.
14      fire-1(x,y);
15    else if ((x<y and (z * z ≤ y) or (current=="South")))
16      ← Condition C3.
17      fire-2(x,y);
18    else
19      fire-3(x,y);
20    input(done);
21  }
22  output("Firing completed.");
23 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.

$C_2 = (x < y) \text{ and } (z * z > y) \text{ and } (\text{prev} == \text{"East"})$					
Test	$x < y$	$z * z > y$	$\text{prev} == \text{"East"}$	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.

$C_3 = (x < y) \text{ and } (z * z \leq y) \text{ or } (\text{current} == \text{"South"})$					
Test	$x < y$	$z * z \leq y$	$\text{current} == \text{"South"}$	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 ($\text{current} == \text{"South"}$) 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](#).

$C_2 = (x < y) \text{ and } (z * z > y) \text{ and } (\text{prev} == \text{"East"})$					
Test	$x < y$	$z * z > y$	$\text{prev} == \text{"East"}$	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.

$C_3 = (x < y) \text{ and } (z * z \leq y) \text{ or } (\text{current} == \text{"South"})$					
Test	$x < y$	$z * z \leq y$	$\text{current} == \text{"South"}$	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 $(\text{prev} == \text{"East"})$ is varied. These two tests demonstrate the independent effect of $(\text{prev} == \text{"East"})$ 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 $(\text{current} == \text{"South"})$ are held constant while $(z * z \leq y)$ is varied. These two tests demonstrate the independent effect of $(z * z \leq y)$ on C_3 . Tests t_1 and t_3 demonstrate the independent effect of $(z * z \leq y)$ on C_3 . However, the independent effect of $(\text{current} == \text{"South"})$ 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 $(\text{prev} == \text{"East"})$ 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 $(z^* z \leq y)$ 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.

$C_2 = (x < y) \text{ and } (z * z > y) \text{ and } (\text{prev} == \text{"East"})$					
Test	$x < y$	$z * z > y$	$\text{prev} == \text{"East"}$	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 ($\text{prev} == \text{"East"}$) 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.

$C_3 = (x < y) \text{ and } (z * z \leq y) \text{ or } (\text{current} == \text{"South"})$					
Test	$x < y$	$z * z \leq y$	$\text{current} == \text{"South"}$	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 ($\text{current} == \text{"South"}$) on C_3 .
t_4	-	-	-	-	Conditions not evaluated.

7.2.11 Minimal MC/DC tests

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.

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.

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 $(x < y \text{ and } \text{done})$ but the condition coded is (done) .

Incorrect Boolean operator: One or more Boolean operators is incorrect. For example, the correct condition is $(x < y \text{ and } \text{done})$ which has been coded as $(x < y \text{ or } \text{done})$.

Mixed type: One or more simple conditions is missing and one or more Boolean operators is incorrect. For example, the correct condition should be $(x < y \text{ and } z * x > y)$

and $d = \text{"South"}$) has been coded as $(x < y \text{ or } z * x \geq y)$.

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	$C_1 \text{ and } C_2 \text{ and } C_3$	$C_1 \text{ and } C_3$	
t	true, true, true	true	true	No
	false, false, false	false	false	No
t	true, true, false	false	false	No
	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 $C' = (C_1 \text{ or } C_2) \text{ and } C_3$. 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	$(C_1 \text{ or } C_2)$ 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	$C = C_1 \text{ or } C_2 \text{ or } C_3$	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

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.

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.

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 $C_1 = \text{false}$ and $C_2 = \text{true}$, or the combination $C_1 = \text{false}$ and $C_2 = \text{false}$, 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 (see [Exercise 7.29](#)).

7.2.15 Tracing test cases to requirements

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*.

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.

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 $(\text{current} = \text{"South"})$.

Note that $R_{1.2}$ is not explicit about what action to take when $(\text{current} = \text{"South"})$ is `false`. Of course, as stated, the requirement does imply that `fire-2` is *not* to be invoked when $(\text{current} = \text{"South"})$ 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 $(\text{current} = \text{"South"})$ 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 R₁. It also stresses the need for the trace-back of all tests to requirements.

7.3 Concepts From Data Flow

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.

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.

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; float z;
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 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 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.

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.

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 derived test set based on data flow will.

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.

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.

```
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:

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.

```
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;
```

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.

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.

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, \dots, 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 $c\text{-use}_i$ denote the set of variables that have a c -use in block i , and $p\text{-use}_i$ 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 $\text{def}_b = \{p, A\}$, $\text{c-use}_b = \{y, z, p, q, x, \text{number}, i\}$, 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 $\neg C$, 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.

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	$\{\}$	$\{z\}$	$\{\}$

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.

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.

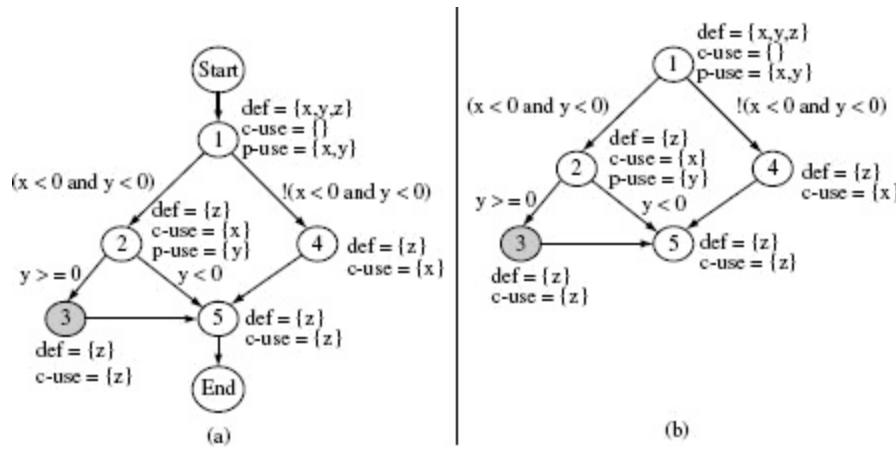


Figure 7.6 Data flow graph of 7.8 (a) start and Exit nodes shown explicitly. No def and use sets are associated with these nodes, (b) Nodes 1 and 5 serve as start and end nodes, respectively. Node 3 is shaded to emphasize that it is unreachable.

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. Similarly, the End node may be omitted in [Figure 7.6](#) and use nodes 1 and 5 as the start and end nodes, respectively.

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 $p = (i, n_1, n_2, \dots, n_k, j)$, $k \geq 0$, that starts at node i , ends at node j , and nodes i and j do not occur along the subpath n_1, n_2, \dots, 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, \dots, 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 .

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.

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 $d_1(\text{count})$ and $u_6(\text{count})$.

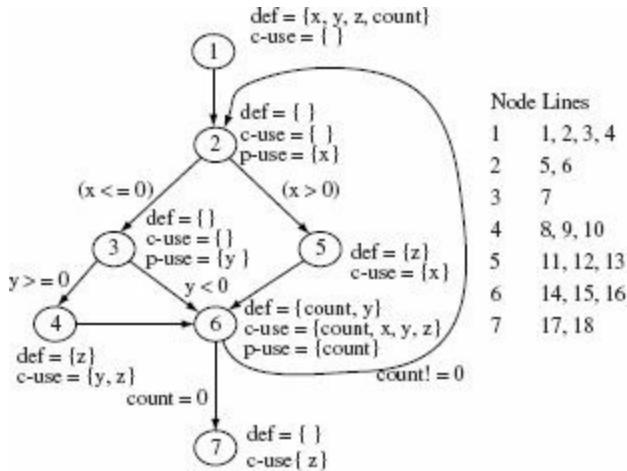


Figure 7.7 Data flow graph of Program [P7.16](#).

Path $q = (6, 2, 5, 6)$ is def-clear with respect to $d_6(\text{count})$ and $d_6(\text{count})$ (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

```

The definition of a variable at some point 11 in a program and its use at some point 12 in the same program constitute a def-sue pair. Note that 11 and 12 could be the same statements in a program.

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 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 $(k, 1)$ such that there is a path from where the variable is defined to location 1; 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 $\text{dcu}(x, 1)$ 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 $\text{dcu}(x, 1)$. Similarly, node 6 is also included in this set. Thus we get $\text{dcu}(x, 1)=\{5, 6\}$.

Next, determine $\text{dpu}(x, 1)$. 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 $\text{dpu}(x, 1)=\{(2, 3), (2, 5)\}$. 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 is 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, \dots, x_{k-1} is a path (n_1, n_2, \dots, 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

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 $\text{dcu}(y, 1) = \{4, 6\}$. We also discovered that $\text{dcu}(z, 1) = \{4, 6, 7\}$. We now ask: *Will the coverage of $\text{dcu}(z, 1)$ imply the coverage of $\text{dcu}(y, 1) = \{4, 6\}$?* 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.

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.

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 $\text{dcu}(y, 1)$ while generating tests to cover all c-uses; $\text{dcu}(y, 1)$ will be covered automatically if we cover $\text{dcu}(z, 1)$.

Arguments, analogous to the ones above, can be used to show that $\text{dcu}(x, 1)$ can be removed from consideration when developing tests to cover all c-uses. This is because $\text{dcu}(x, 1)$ will be covered when $\text{dcu}(z, 5)$ is covered. Continuing in this manner, one can show that $\text{dcu}(\text{count}, 1)$ 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 χ SUDS, 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

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 $DE(n)$.

A data environment is the set of all variables whose definitions are live at a node in a data flow graph.

Let $X(n) = \{x_1, x_2, \dots, x_k\}$ 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_i^j denote the j th definition of x_i in F . An *elementary data context* of node n , denoted as $EDC(n)$, is the set of definitions $(x_1^{i_1}, x_2^{i_2}, \dots, x_k^{i_k})$.

Contexts are often represented as sets of definitions, e.g., $\{(x_1^1, x_2^2, \dots, x_k^k)\}$.

≥ 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_j th 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	$\{ (d_1(z), d_1(y), d_1(z), d_1(\text{count})), (d_1(z), d_1(y), d_5(z), d_1(\text{count})), (d_1(z), d_1(y), d_4(z), d_1(\text{count})), (d_1(z), d_6(y), d_5(z), d_6(\text{count})), (d_1(z), d_6(y), d_4(z), d_6(\text{count})) \}$
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 $(1, 2, 3, 4, 6, 2, 3, 4, 6, 7)$, $d_4(z)$ 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 $d_6(y)$, $d_6(\text{count})$, 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 $d_6(\text{count})$. 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 $DC(6)$.

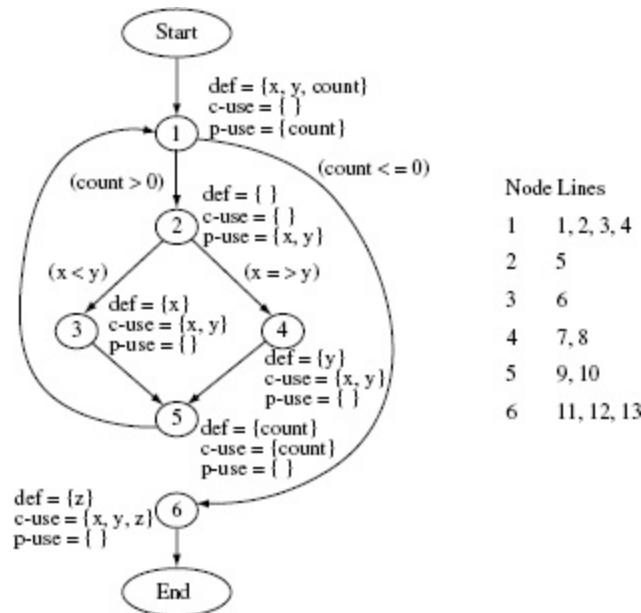


Figure 7.8 Data flow graph of Program P7.17 to illustrate ordered data context.

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
```

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 ,

$$(n_{i_1}, n_{i_2}, \dots, n_{i_{m-1}}, n_{i_m})$$

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 $1 \leq j \leq (m - 1)$.

Test adequacy criteria based on data flows can be defined using a data flow graph.

Let CU and PU denote, respectively, the total number of c-uses and p-uses for all variable definitions in P . Let $v = \{v_1, v_2, \dots, v_n\}$ 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.

$$\begin{aligned} CU &= \sum_{i=1}^n \sum_{j=1}^{d_i} |\text{dcu}(v_i, n_j)| \\ PU &= \sum_{i=1}^n \sum_{j=1}^{d_i} |\text{dpu}(v_i, n_j)| \end{aligned}$$

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

Let z be a node in $\text{dcu}(x, q)$, 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:

$p = (n_1, n_{i_1}, \dots, n_{i_l}, n_{i_{l+1}}, \dots, n_{i_m}, n_{i_{m+1}} \dots n_k)$, where $2 \leq i_j < k$ 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 $(n_{i_l}, n_{i_{l+1}}, \dots, n_{i_m})$ is a def-clear path from node q to node z . All c-uses of variable x are considered covered if each node in $\text{dcu}(x, q)$ 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.

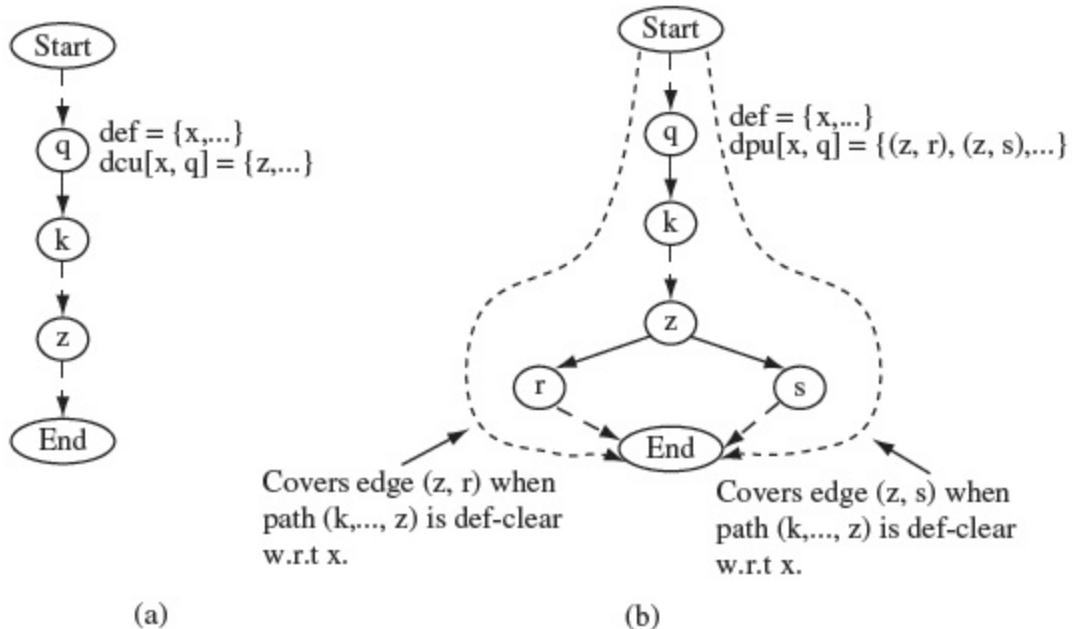


Figure 7.9 Paths for c-use and p-use coverage. Dotted arrows connecting two nodes indicate a path along which there might be additional nodes not shown. (a) Path $(\text{Start}, \dots, q, k, \dots, z, \dots, \text{End})$ covers the c-use at node z of x defined at node q given that (k, \dots, z) is def-clear with respect to x . (b) The p-use of x at node z is covered

when paths $(\text{Start}, \dots, q, k, \dots, z, r, \dots, \text{End})$ and $(\text{Start}, \dots, q, k, \dots, z, s, \dots, \text{End})$, shown by dotted lines, have been traversed, and (k, \dots, z) are def-clear with respect to x .

A c-use of a variable x defined at location 11 and used at location 12 at is considered covered when control flows along a path from 11 to 12 without any redefinition of 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 $\text{dcu}(z, 5)$, i.e. the desired test must cover the c-use of the definition of x at node 1. Consider the following test case.

$t_c : \langle x = 5, y = -1, \text{count} = 1 \rangle$

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 $\text{dcu}(z, 6)$.

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 $\text{dpu}(x, q)$, 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 .

$(n_1, n_{i_2}, \dots, n_{i_1}, n_{i_{l+1}}, \dots, n_{i_m}, n_{i_{m+1}}, \dots, n_k)$ where $2 \leq i_j < k$ for $1 \leq j \leq k$

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 $(n_{i_l}, n_{i_{l+1}}, \dots, n_{i_m}, n_{i_{m+1}})$ 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 $(n_{i_l}, n_{i_{l+1}}, \dots, n_{i_m}, n_{i_{m+1}})$ is a def-clear path with respect to x .

A p-use of a variable x defined at location 11 and used at location 12 in a predicate at location 12 is considered covered when control flows along a path from 11 to 12 and then to each of the destinations of the predicate. Of course, each p-use coverage requires the traversal of at least two paths.

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 $y \geq 0$ is covered if taken, though the p-use is yet to be covered. Subsequently, control should take the edge labeled $y < 0$ during the same or in some other execution. Consider the following test case designed to achieve the coverage objective.

$t_p : \langle x = -2, y = -1, \text{count} = 3 \rangle$

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 .

$p = \langle 1, 2, 3, 6_1, 2, 3_1, 4_1, 6_2, 2, 3_2, 6_3, 7 \rangle$

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 $\langle 1, 2, 3, 6_1 \rangle$ is traversed. This value of y is used in the decision at node 3 when $\langle 6_1, 2, 3_1, 4_1 \rangle$ is traversed. Note that edge $(3, 4)$ is now covered as there is a def-clear path $\langle 2, 3 \rangle$ from node 6 to edge $(3, 4)$.

Next, path $\langle 6_2, 2, 3_2, 6_3, 7 \rangle$ 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 $\langle 2, 3_2, 6_2 \rangle$. The control now moves to node 7 and the program execution terminates. Test t_p has successfully covered $\text{dpu}(y, 6)$. 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 $C=(n_1, n_2, \dots, n_k)$ 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:

(Start, $\dots, n_1, p_1, n_2, \dots, p_{k-1}n_k, \dots, \text{End}$)

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:

(Start, $\dots, n_1, p_1, n_2, \dots, p_{k-1}n_k, r, \dots, \text{End}$)
 (Start, $\dots, n_1, p'_1, n_2, \dots, p'_{k-1}n_k, s, \dots, \text{End}$)

where p_i and p' s 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 $(d_1(x), u_5(x), d_5(z), d_7(z))$. The following test covers C .

$$T = \{t : \langle x=3, y=2, \text{count}=1 \rangle\}$$

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:

$$\begin{aligned}\lambda_1 &= (1, p_1, 4, p_2, 6, 2) \\ \lambda_2 &= (1, p'_1, 4, p'_2, 6, 7)\end{aligned}$$

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 :

$$T = \left\{ \begin{array}{l} t_1: \langle x=-5, y=2, \text{count}=1 \rangle \\ t_2: \langle x=-5, y=2, \text{count}=2 \rangle \end{array} \right\}$$

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.

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.

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.

7.4.6 Infeasible c- and 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.

An infeasible path in a program may lead to infeasible c- or p-uses.

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](#).

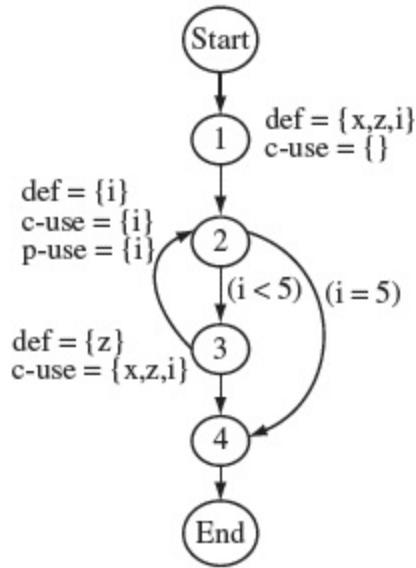


Figure 7.10 Data flow graph of P7.18.

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.

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 .

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 $OEDC(k)$.

A test set is considered adequate with respect to the data context criterion if it covers all the feasible data contexts.

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:

$$\frac{EDC_e}{EDC - EDC_i}$$

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 contexts 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 adequacy of test T given below against the elementary data context coverage criteria.

$$T = \left\{ \begin{array}{l} t_1: \langle x=-2 \ y=2 \ count=1 \rangle \\ t_2: \langle x=-2 \ y=-2 \ count=1 \rangle \\ t_3: \langle x=2 \ y=2 \ count=1 \rangle \\ t_4: \langle x=2 \ y=2 \ count=2 \rangle \end{array} \right\}$$

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 <i>EDC</i> coverage
t_1	(1, 2, 3, 4, 6, 7)	4: $(d_1(y), d_1(z))$ 6: $(d_1(x), d_1(y), d_4(z), d_1(\text{count}))$ 7: $(d_4(z))$	$\frac{3}{12} = 0.25$
t_2	(1, 2, 3, 6, 7)	6: $(d_1(x), d_1(y), d_1(z), d_1(\text{count}))$ 7: $(d_1(z))$	$\frac{5}{12} = 0.42$
t_3	(1, 2, 5, 6, 7)	5: $(d_1(x))$ 6: $(d_1(x), d_1(y), d_5(z), d_1(\text{count}))$	$\frac{8}{12} = 0.67$
t_4	(1, 2, 5, 6, 2, 5, 6, 7)	7: $(d_5(z))$ 5*: $(d_5(x))$ 6*: $(d_1(x), d_1(y), d_5(z), d_1(\text{count}))$ 5*: $(d_5(x))$ 6: $(d_1(x), d_6(y), d_5(z), d_6(\text{count}))$ 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. 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).

$$T = \{t_1 : x = -2, y = 2, \text{count} = 2 >, t_2 : < x = 2, y = 2, \text{count} = 1 >\}$$

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.

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.

$$T = \left\{ \begin{array}{lll} t_1: & \langle x=-2, y=2, count=1 \rangle \\ t_2: & \langle x=-2, y=-2, count=1 \rangle \\ t_3: & \langle x=2, y=2, count=1 \rangle \\ t_4: & \langle x=2, y=2, count=2 \rangle \end{array} \right\}$$

Execution of [P7.16](#) against T produces the following values of z :

$$\begin{aligned} P7.16(t_1) &= 1.0 \\ P7.16(t_2) &= 0.0 \\ P7.16(t_3) &= 0.5 \\ P7.16(t_4) &= 0.5 \end{aligned}$$

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.

$$t_e : \langle x = -2, y = 2, count = 2 \rangle$$

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:

$$T = \left\{ \begin{array}{l} t_1: \langle x=5 \ y=-2 \ z=1 \rangle \\ t_2: \langle x=5 \ y=2 \ z=-1 \rangle \end{array} \right\}$$

However, T does not cover the c-use ($d_3(x)$, $u_{11}(x)$). 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 ($d_3(x)$, $u_{11}(x)$) 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.

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.

Example 7.46 Program P7.20 takes two inputs x and y , and computes z .

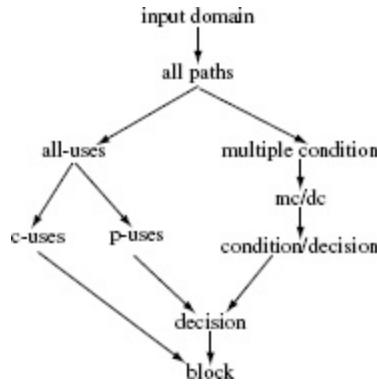


Figure 7.11 The subsumes relationship amongst various control and data flow based test adequacy criteria, $x \rightarrow y$ indicates that x subsumes y . See the bibliography section for citations to research that points to the assumptions that need to be satisfied for the relationship to hold.

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.

$$T = \{ t: \langle x = 5 \ y = -2 \rangle \}$$

7.7 Structural and Functional 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.

Measuring code coverage while testing, and enhancing tests based on such measurements, is often referred to as “white box” or “structural” testing.

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 requirement $R_{39.3}$.
- 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 as JUnit. However, doing so for very large and 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.

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.

It is often easier to determine code coverage for non-embedded versus embedded 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.

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.

Even for large systems, coverage measurement can be made possible and effective through the use of an incremental approach.

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 encapsulates 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 generates 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 χ SUDS 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 $Cond = (A \text{ and } B) \text{ or } (A \text{ and } C) \text{ or } (B \text{ and } C)$ 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 $f_2(x, y, z) = x * y * z$.

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	$f_1(x, y, z)$

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?

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 programs to show that in general this claim is not true.

7.10 Let A , B , and C denote three Booleans. Let $C' = (A \text{ and } B) \text{ or } (A \text{ and } C) \text{ or } (B \text{ and } C)$ 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 $C = C_1 \text{ or } C_2 \text{ or } C_3$, given MC/DC adequate tests for $C = C_1 \text{ or } C_2$ and (b) derive MC/DC adequate tests for $C = C_1 \text{ xor } C_2 \text{ xor } C_3$, given MC/DC adequate tests for $C = C_1 \text{ xor } C_2$.
- 7.16 Extend the procedure in [Section 7.2.9](#) to derive MC/DC adequate tests for $C = C_1 \text{ and } C_2 \dots \text{ and } C_{n-1} \text{ and } C_n$, given MC/DC adequate tests for $C = C_1 \text{ and } C_2 \dots \text{ and } C_{n-1}$.
- 7.17 Use the procedures derived in [Exercises 7.15](#) and [7.16](#) to construct an MC/DC adequate test for (a) $(C_1 \text{ and } C_2) \text{ or } C_3$, (b) $(C_1 \text{ or } C_2) \text{ and } C_3$, and (c) $(C_1 \text{ or } C_2) \text{ xor } C_3$.
- 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 , ..., 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 $C = C_1 \text{ and } C_2 \text{ and } \dots \text{ and } C_n$, suppose that C has been coded as $C' = C_1 \text{ and } C_2 \text{ and } \dots \text{ and } C_{i-1} \text{ and } C_{i+1} \dots \text{ and } C_n$; 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 $C = C_1 \text{ or } C_2 \text{ or } \dots \text{ or } C_n$ and $C' = C_1 \text{ or } C_2 \text{ or } \dots \text{ or } C_{i-1} \text{ or } C_{i+1} \dots \text{ or } C_n$.

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:

$$D = (A \text{ and } B) \text{ or } (A \text{ and } C) \text{ or } (B \text{ and } C)$$

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 [Figure 7.7](#) 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 , $0 < e \leq 1$.

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 $\text{foo1}()$, $\text{foo2}()$, $\text{foo3}()$, $\text{foo4}()$, $\text{P1}()$, and $\text{P2}()$ 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.”

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 object-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*.

Program mutation can be used as a white-box as well as a black-box technique.

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.

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.

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.

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.

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.

A program P under test is mutated by applying a mutation operator to P that results in a mutant.

Example 8.1 Consider the following simple program.

Program P8.1

```
1 begin
2   int x, y;
3   input (x, y);
4   if (x<y)
5     output (x+y);
6   else
7     output (x*y);
8 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](#)

```
1 begin
2   int x, y;
3   input (x, y);
4   if (x≤y)      ← Mutated statement
5   then
6     output (x+y);
7   else
8     output (x*y);
9 end
```

Mutant M2 of Program [P8.1](#)

```
1 begin
2   int x, y;
3   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 *n*th 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 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](#).

For a given program P the number of higher order mutants could be significantly larger than the number of first order mutants.

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

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.

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.

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

```
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=veryHigh;  
15     return(danger);  
16 }
```

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;  
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; ← Mutated statement  
13    if (highCount==2) danger=high;  
14    if (highCount==3) danger=veryHigh;  
15    return(danger) ;  
16 }
```

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*.

In strong mutation only the external outputs of program P and its mutant M are compared, to determine whether or not the two have differed. In weak mutation the comparison could be carried out at intermediate stages during the execution and among internal state variables of P and M.

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.

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 :

```
 $t : < \maxTemp = 1200, \currentTemp = [1250, 1389, 1127] >$ 
```

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.

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?

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 ?” ”`

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.

Some times a program and its mutant might be equivalent though one is preferred over the other for reasons such as performance or maintainability.

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.

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.

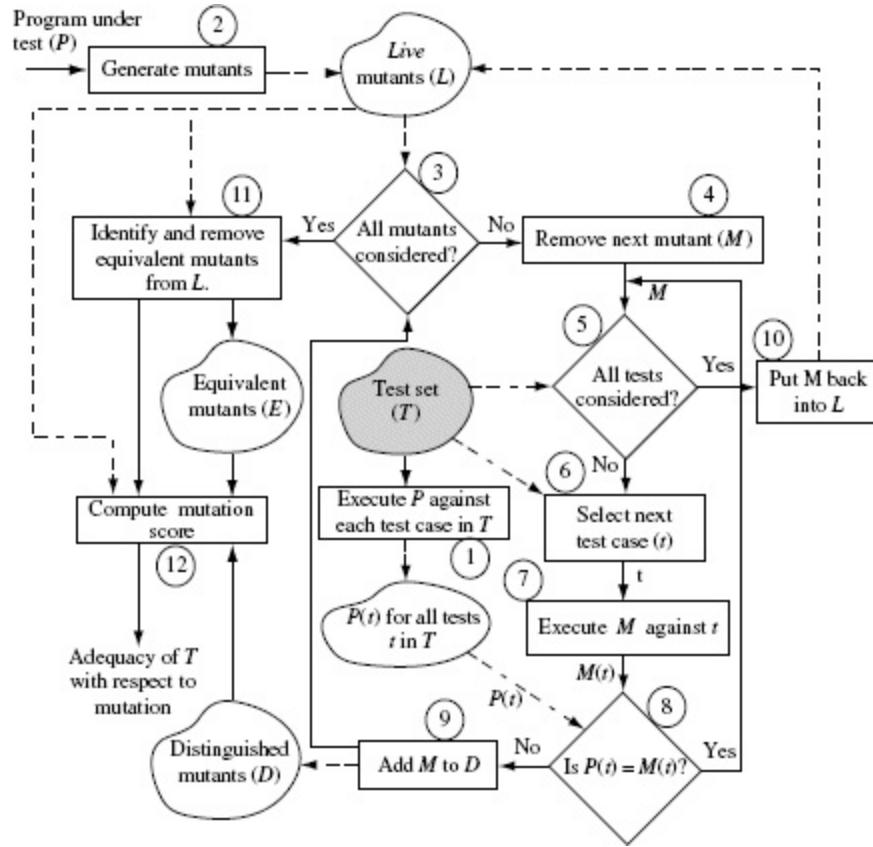


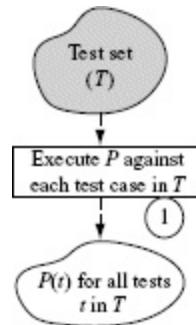
Figure 8.1 A procedure used in the assessment of the adequacy of a test set using mutation. Solid lines point to the next process step. Dashed lines indicate data transfer between a data repository and a process step. L, D, and E denote, respectively, the set of live, distinguished, and equivalent mutants 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.

$$T_P = \left\{ \begin{array}{l} t_1: <x=0, y=0>, \\ t_2: <x=0, y=1>, \\ t_3: <x=1, y=0>, \\ t_4: <x=-1, y=-2> \end{array} \right\}$$

The database of $P(t)$ for all $t \in T_P$, is tabulated below.

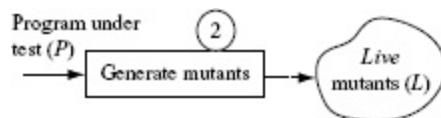
Test case (t)	Expected output $f(x, y)$	Observed output $P(t)$
t_1	0	0
t_2	1	1
t_3	0	0
t_4	1	2

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

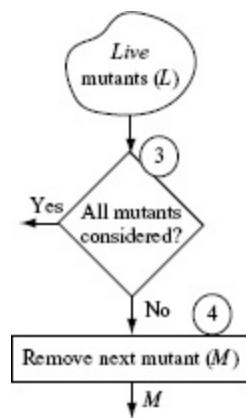
In the table above, notice that we have not mutated the declaration, input, then, and else statements. The reason for not mutating these statements is given later in [Section 8.4](#). Of course, markers begin and end are not mutated.

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

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 .

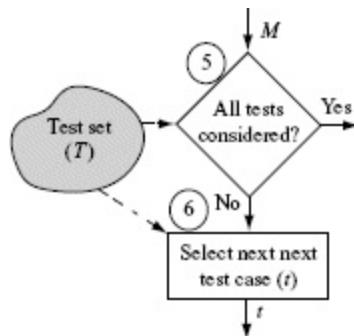
After having created mutants, each mutant is selected and executed against tests in T until it shows a behavior different from that of the program under test or it has been executed against all tests.



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.



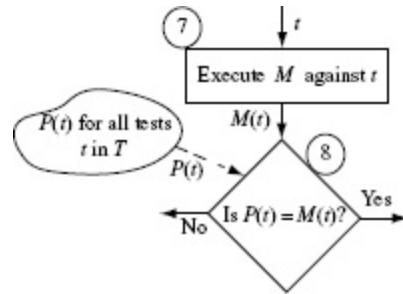
Example 8.10 Now that we have selected M_1 , we need to check if can be distinguished by T from P . In Step 5 we select the next test. Again, any of the tests in T_P , against which M_1 has not been executed so far, can be selected. Let us select $t_1: < x = 0, y = 0 >$.

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 M_1 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 M_1 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 }

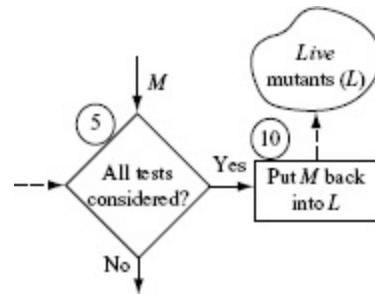
Program	t_1	t_2	t_3	t_4	D
$M_3(t)$	0	2*	NE	NE	{ M_1, M_3 }
$M_4(t)$	0	2*	NE	NE	{ M_1, M_3, M_4 }
$M_5(t)$	0	-1*	NE	NE	{ M_1, M_3, M_4, M_5 }
$M_6(t)$	0	1	0	0*	{ M_1, M_3, M_4, M_5, M_6 }
$M_7(t)$	0	1	1*	NE	{ $M_1, M_3, M_4, M_5, M_6, M_7$ }
$M_8(t)$	U*	NE	NE	NE	{ $M_1, M_3, M_4, M_5, M_6, M_7, M_8$ }

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 .

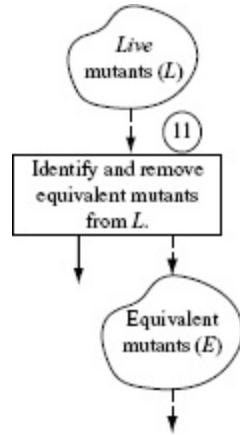


Example 8.13 In our example, there is only one mutant, M_2 , that could not be distinguished by T_P . M_2 is returned to the set of live mutants in [Step 10](#). Notice, however, that M_2 will not be selected again for classification as it has been selected once.

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.



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.

To obtain an accurate mutation score for test T , one needs to determine which of the live mutants are equivalent to the parent program.

$$f_P(x, y) = \begin{cases} x + y & \text{if } x < y, \\ x * y & \text{otherwise.} \end{cases}$$

$$g_{M_2} = \begin{cases} x + y & \text{if } x < y + 1, \\ x * y & \text{otherwise.} \end{cases}$$

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.

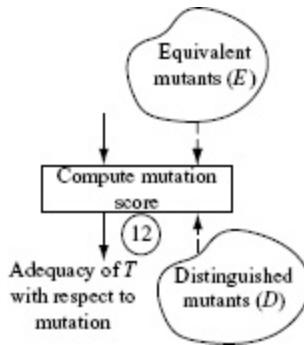
Step 12: Computation of mutation score

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.

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.

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.

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

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.

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.

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 danger_P and danger_M denote the values of variable *danger* soon after the execution of line 14. The state infection condition is satisfied when $\text{danger}_P \neq \text{danger}_M$. Any test case for which `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.

$t : < \text{currentTemp} = [20, 34, 29], \text{maxTemp} = 18 >$

For test case t , we get $P(t) = \text{veryHigh}$ and $M(t) = \text{none}$ 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 of the corresponding atomic weights. A global constant `maxSize` is the size the two arrays. Consider a mutant of [P8.3](#) obtained by replacing the loop condition by `index ≤ size`.

Program P8.3

```
1  String findElement (name, atWeight, int size,
2      float-w) {
3      String name [maxSize]; float atWeight [maxSize], w;
4      int index=0;
5      while (index<size) {
6          if (atWeight [index]>w)
7              return(name [index]);
8          index=index+1;
9      }
10     return("None");
11 }
```

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.

```
t1: < 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 *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.

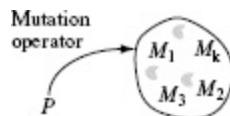


Figure 8.2 A set of k mutants M_1, M_2, \dots, M_k of P generated by applying a mutation operator. The number of mutants generated, k , depends on P and the mutation operator; k could be 0.

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 $\text{abs}(e)$. 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	<code>if(x<y)</code>	<code>if(abs(x)<y)</code>
		<code>if(x<abs(y))</code>

Line 5	$\text{output}(x+y)$	$\text{output}(\text{abs}(x)+y)$
		$\text{output}(x+\text{abs}(y))$
		$\text{output}(\text{abs}(x+y))$
Line 7	$\text{output}(x*y)$	$\text{output}(\text{abs}(x)*y)$
		$\text{output}(x*\text{abs}(y))$
		$\text{output}(\text{abs}(x*y))$

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.

Mutation operators can be categorized with respect to, for example, the generic syntactic element in the program on which they are applicable.

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 (\rightarrow) as *mutates to*. Also, note that in the first entry, constant 1 has been changed to 3, as constant 3 appears in another statement ($y=y^*3$).

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	$x=x+1; \rightarrow x=+3;$ $y=y^*3; \rightarrow y=y^*1;$
Operator mutations	Incorrect operator	$if(x < y) \rightarrow if(x \leq y)$ $x++ \rightarrow ++x$
Statement mutation	Incorrectly placed statement	

		$z=x+1; \rightarrow \text{delete}$ $z=x+1; \rightarrow \text{break}$ $\text{break;} \rightarrow z=x+1;$
Variable mutations	Incorrectly used variable	$z=x+1; \rightarrow z=y+1;$ $z=x+y; \rightarrow z=\text{abs}(x)+y;$

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 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 { $<$, \leq , $>$, \geq , \neq , $=$ }.

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 incorrectly place declarations often 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 $=$, \neq , $>$, \leq , and \geq . 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 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 $P \in P_L$, 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.

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.

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.

8.6 Founding Principles of Mutation Testing

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.

Appropriate use of mutation testing can lead to close-to-correct programs.

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.

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.

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.

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.

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.

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 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 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.

The problem of 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.

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 not 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

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.

Despite the simplicity of the first order mutants, mutation testing can reveal nearly all types of faults in a program.

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.

```

1  int misCond(data, link, F) {
2  int data[3], link [3], F;      ← Inputs.
3  int sum;          ← Output.
4  int L;           ← Local variable.
5  sum=0;
6  L=F;
7  if(L≠ -1){       ← Part of the condition is missing.
8    while (L≠ -1){
9      if(data[L] ≠ 0) sum=sum+data[L];   ← Redundant
                                         condition.
10     L=link[L];
11   };
12 }
13 return(sum);
14 }
```

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

```
((L != -1) and (data[L] != 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];
```

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.

A mutant M is considered “error-revealing” when a test that distinguishes M also reveals an error in its parent program.

We will now determine a test case that distinguishes M from `misCond`. Let C_P and C_M denote the two conditions $\text{data}(L) \neq 0$ and $\text{data}(F) \neq 0$, respectively. Let SUM_P and 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*: $\text{SUM}_P \neq \text{SUM}_M$.

We must have $L = F \neq -1$ 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 $\text{link}(F) \neq -1$. Any one of the following two conditions could be true during the second or subsequent loop traversals.

$$\text{if } \text{data}(F) \neq 0 \text{ then } \text{data}(L) = 0 \quad (8.1)$$

$$\text{if } \text{data}(F) = 0 \text{ then } \text{data}(L) \neq 0 \quad (8.2)$$

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 $\text{misCond}(t) \neq M(t)$:

$$\begin{aligned} & F \neq -1 \\ & \text{link}(F) \neq -1 \\ & \text{data}(F) = 0 \\ & \sum_j \text{data}(j) \neq 0 \text{ where } j \text{ varies from } \text{link}(F) \text{ to the index of the} \\ & \quad \text{last element in the sequence.} \end{aligned}$$

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	\rightarrow	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 . 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 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 revealing* (ε) for program P if and only if $\forall t \in D$ 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 M is said to be of type *error hinting* (H), if and only if $P \equiv M$ and $P_c \neq M$.

A mutant M is said to be of type *reliability indicating* (R) if and only if $P(t) \neq M(t)$ for some $t \in D$ 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 $S_\varepsilon \cap S_H = \emptyset$ and $S_H \cap S_R = \emptyset$. A test case that distinguishes a mutant either reveals the error in which case it belongs to S_ε , or else it does not reveal the error in which case it belongs to S_R . Thus $S_\varepsilon \cap S_R = \emptyset$.

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 mutation operators known for C. Josè Maldonado's research group at the University of São Carlos at São 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 , cS denotes the condition controlling the execution of S . If S is a `for` statement, then eS 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 iS denotes the initialization expression executed exactly once for each execution of S . If the controlling condition is missing, then cS 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 cS . If S denotes a `for` statement, then in an execution sequence we shall refer to the execution of S by one reference to iS , one or more references to cS , and zero or more references to eS . 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,

iS : $m=0, n=0$

cS : `isdigit(s[i])`, and

eS : `i++`.

If S denotes the following `for` statement,

```
for (; ;){
    ;
}
```

then we have,

iS : (the null expression-statement),

cS : `true`, and

eS : (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 $X'_j, 1 \leq j \leq i$ denote the sequence $X_j X_{j+1} \dots X_{i-1} X_i$. Let

$S_L = S'_1, l \geq 1$ 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 $E^s(f, t)$; $E^s(f, t)$; $E^l(f, t)$ denotes the last statement of the execution sequence of f . If f is terminating, $E^l(f, t) = \text{return}$.

Let $E_1 = S'_i$ and $E_2 = Q'_k$ be two execution sequences. We say that E_1 and E_2 are *identical* if and only if $i = k$, $j = l$, and $S_q = Q_q$, $i \leq q \leq j$. 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[])
{
    int n;
    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    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: $S_1^i S_2^c S_3^c S_2^c S_3^c S_4 S_5 S_6$. $S_1^i S_2^c$ 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 $E(trim, t)$. $S_1^i S_2^c S_3^c S_2^c S_6$ 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:

$$E_1 |_c E_2 = \begin{cases} E_1 & \text{if } c \text{ is false,} \\ E_1 \circ E_2 & \text{otherwise.} \end{cases}$$

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 $|_{\text{true}}$. \circ associates from left to right and $|_c$ associates from right to left. Thus, we have:

$$\begin{aligned} E_1 \circ E_2 \circ E_3 &= (E_1 \circ E_2) \circ E_3 \\ E_1 |_{c_1} E_2 |_{c_2} E_3 &= E_1 |_{c_1} (E_2 |_{c_2} E_3) \end{aligned}$$

$E(f, *)$ ($E(S, *)$) 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 $E(\{ \}, t) |_c E(S, t) = |_c E(S, t) = \begin{cases} \text{null sequence} & \text{if } c \text{ is false,} \\ E(S, t) & \text{otherwise.} \end{cases}$

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 $S = \text{while } (c) S'_1$, then

$$E(S, t) = |_c (E(S'_1, *) \circ E(S, *)).$$

If $RLS(S) = S'_1$, $n > 1$, and $S_i = \text{continue}$, $1 \leq i \leq n$, then

$$E(S, t) = |_c E(S'_1, *) \circ (|_{(E(S'_1, *) \neq \text{continue})} E(S'_{i+1}, *)) \circ E(S, *).$$

If $RLS(S) = S'_1$, $n > 1$, and $S_i = \text{break}$, $1 \leq i \leq n$, then

$$E(S, t) = |_c E(S'_1, *) \circ (|_{(E(S'_1, *) \neq \text{break})} (E(S'_{i+1}, *) \circ E(S, *))).$$

R 8 If $S = \text{do } S_1 \text{while}(c)$; then

$$E(S, t) = E(S_1, t) |_c E(S, *).$$

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 $S = \text{if } (c) S_1$, then

$$E(S, t) = |_c E(S_1, *).$$

R 10 If $S = \text{if } (c) S_1 \text{ else: } S_2$, 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 $E(S_3, *) = {}^c S_3$. If $n = 2$, then $E(S_3, *) = {}^c S_3 S_4$. Similarly, for S_2 , which is a `for` statement, we get $E(S_2, *) = {}^i S_2 {}^c S_2 {}^c S_3 {}^c S_3 S_4$. 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

$$E(trim', t) = S_1^i S_2^c S_2^c S_3^e S_2^c S_3^e S_2^c S_3^e S_3^e S_4^e S_2^c S_3^e S_4^e S_2^c S_5^e S_6.$$

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.

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 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 .

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

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 $ps->a$.

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.

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;
};

struct rect screen;
struct key {
    char *word;
    int count;
} keytab [ NKEYS ];

```

F2

The global sets corresponding to the above declarations are

$$\begin{aligned}
G_f &= \{i, j, c, d, r, s, p, q, screen, keytab\}, \\
GS_f &= \{i, j, c, d, r, s\}, \\
GP_f &= \{p\}, \\
GT_f &= \{screen\}, \text{ and} \\
GA_f &= \{q, keytab\}.
\end{aligned}$$

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 (*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](#).

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

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.

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  
k = ui + *p  
k = j +null
```

M1

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.

Pointer constant, `null`, is not mutated. Left operands of assignment, `++` and `--` operators are also not mutated.

Null is 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	$a += b \rightarrow a -= b$
OAAN	Arithmetic	Arithmetic	$a + b \rightarrow a * b$
OBBA	Bitwise assignment	Bitwise assignment	$a \&= b \rightarrow a = b$
OBBN	Bitwise	Bitwise	$a \& b \rightarrow a b$
OLLN	Logical	Logical	$a \&& b \rightarrow a b$
ORRN	Relational	Relational	$a < b \rightarrow a \leq b$
OSSA	Shift assignment	Shift assignment	$a <<= b \rightarrow a >>= b$
OSSN	Shift	Shift	$a << b \rightarrow a >> b$

[†] Read $X \rightarrow Y$ 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	$a += b \rightarrow a = b$
OAEA	Arithmetic assignment	Plain assignment	$a += b \rightarrow a = b$
OABN	Arithmetic	Bitwise	$a + b \rightarrow a \& b$
OALN	Arithmetic	Logical	$a + b \rightarrow a \&& b$
OARN	Arithmetic	Relational	$a + b \rightarrow a < b$
OASA	Arithmetic assignment	Shift assignment	$a += b \rightarrow a <<= b$
OASN	Arithmetic	Shift	$a + b \rightarrow a << b$
OBAA	Bitwise assignment	Arithmetic assignment	$a = b \rightarrow a += b$
OBAN	Bitwise	Arithmetic	$a \& b \rightarrow a + b$
OBEA	Bitwise assignment	Plain assignment	$a \&= b \rightarrow a = b$
OBLN	Bitwise	Logical	$a \& b \rightarrow a \&& b$
OBRN	Bitwise	Relational	$a \& b \rightarrow a < b$
OBSA	Bitwise assignment	Shift assignment	$a \&= b \rightarrow a <<= b$
OBSN	Bitwise	Shift	$a \& b \rightarrow a << b$

[†] Read $X \rightarrow Y$ as “X gets mutated to Y.”

Table 8.5 Domain and Range of mutation operators in **Oior**: plain, logical, and relational.

Name	Domain	Range	Example
OEAA	Plain assignment	Arithmetic assignment	$a = b \rightarrow a += b$
OEBA	Plain assignment	Bitwise assignment	$a = b \rightarrow a \&= b$
OESA	Plain assignment	Shift assignment	$a = b \rightarrow a <<= b$
OLAN	Logical	Arithmetic	$a \&\& b \rightarrow a + b$
OLBN	Logical	Bitwise	$a \&\& b \rightarrow a \& b$
OLRN	Logical	Relational	$a \&\& b \rightarrow a < b$
OLSN	Logical	Shift	$a \&\& b \rightarrow a << b$
ORAN	Relational	Arithmetic	$a < b \rightarrow a + b$
ORBН	Relational	Bitwise	$a < b \rightarrow a \& b$
ORLN	Relational	Logical	$a < b \rightarrow a \&\& b$
ORSN	Relational	Shift	$a < b \rightarrow a << b$
OSAA	Shift assignment	Arithmetic assignment	$a <= b \rightarrow a += b$
OSAN	Shift	Arithmetic	$a << b \rightarrow a + b$
OSBA	Shift assignment	Bitwise assignment	$a << b \rightarrow a \mid= b$
OSBN	Shift	Bitwise	$a << b \rightarrow a \& b$
OSEA	Shift assignment	Plain assignment	$a <= b \rightarrow a = b$
OSLN	Shift	Logical	$a << b \rightarrow a \&\& b$
OSRN	Shift	Relational	$a << b \rightarrow a < b$

[†] Read $X \rightarrow Y$ 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 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: $x op \sim y$, $\sim x op y$, and $\sim(x op y)$. **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 $\overbrace{* * \dots *}_n x op$. **OIPM** systematically mutates this reference to the following references:

$$\begin{array}{ccc}
 \overbrace{* * \dots *}_{n-1} & (*x) & op \\
 \overbrace{* * \dots *}_{n-1} & op & (*x) \\
 \overbrace{* * \dots *}_{n-2} & (**x) & op \\
 \overbrace{* * \dots *}_{n-2} & op & (**x) \\
 \vdots & \vdots & \vdots \\
 (\overbrace{ * \dots *}_{n-1} x)op & & \\
 * & op & (\overbrace{* * \dots *}_{n-1} x) \\
 (\overbrace{* * \dots *}_{n} x)op & & \\
 op & (\overbrace{* * \dots *}_{n} x)
 \end{array}$$

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, \leftrightarrow may be read as “gets mutated to.” All entities to the left of \leftrightarrow 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  $\leftrightarrow$  signed char unsigned char  
int* float*
```

```
int  $\leftrightarrow$  signed int unsigned int  
short int long int  
signed long int signed long int  
float* char*
```

```
float  $\leftrightarrow$  double long double  
int* char*
```

```
double  $\leftrightarrow$  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))
```

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.

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 SGLR	do-while goto	do-while replacement by while goto label replacement
SMVB	Statement	Move brace up and down

SRSR	<code>return</code>	<code>return replacement</code>
SSDL	Statement	Statement deletion
SSOM	Statement	Sequence Operator Mutation
STRI	<code>if</code> Statement	Trap on <code>if</code> condition
STRP	Statement	Trap on statement execution
SMTC	Iterative statements	n -trip continue
SSWM	<code>switch</code> statement	<code>switch</code> statement mutation
SMTT	Iterative statement	n -trip trap
SWDD	<code>while</code>	<code>while</code> replacement by <code>do-while</code>

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)
        y -= x;
    else
        x -= y;
}

```

F3

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();
while (x != y)
{
    trap_on_statement();
}
while (x != y)
{
    if (x < y)
        trap_on_statement();
    else
        x -= y;
}
while (x != y)
{
    if (x < y)
        y -= x;
    else
        trap_on_statement();
}

```

M2

M3

M4

M5

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:

<i>v</i> = <i>e</i>	
if (trap_on_true (<i>v</i>)) <i>S</i>	M6
<i>v</i> = <i>e</i>	
if (trap_on_false (<i>v</i>)) <i>S</i>	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)
{
}
while (x != y)
{
    if (x < y)
        ;
    else
        x -= y;
}
while (x != y)
{
    if (x < y)
        y -= x;
    else
        ;
}
```

M9

M10

M11

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 follow 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 $E(f, t) = s^m R$, then there are $m + 1$ possible suffixes of $E(f, t)$. These are enumerated below:

$$\begin{aligned} & s_1 s_2 \dots s_{m-1} s_m R \\ & s_2 \dots s_{m-1} s_m R \\ & s_{m-1} s_m R \\ & \vdots \\ & R \end{aligned}$$

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, \dots, 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 $E^s(f, i)$ 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 Ritchie'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++)
            ;
        if (k>0 && t[k]=='\0')
            return i;
    }
    return -1;
}
```

F4

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. */
    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. */
        if (k>0 && t[k]=='\0')
            return i;
    }
    return -1;
}
```

M12

M13

Note that both [M12](#) and [M13](#) generate the shortest possible PMES for *f*.

goto Label Replacement

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 , ..., 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 $l_1, l_2, \dots, l_{i-1}, l_{i+1}, \dots, l_n$. If $n=1$, no mutants are generated by **SGLR**.

The replacement of a label in a jump statement models an erroneous jump destination.

continue *Replacement by break*

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.

The incorrect use of the continue and break statements is modeled by a few mutation operators.

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 n^{th} 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;  
}  
while (--lim>0 && (c=getchar()) != EOF && c != '\n')                                         M14
```

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';  
} while ((n /= 10) > 0);
```

F6

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_nth_loop_iteration(n)`. When the guard is evaluated the n^{th} 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 ])
        swap(v, ++last, i);
```

F7

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 ])
            swap(v, ++last, i);
    }
```

M16

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_nth_loop_iteration(n)`. During the first n iterations of the loop, `false_after_nth_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 ])
      swap (v, ++last, i);
  }

```

M17

The **SMT_C** 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 $\text{RAP}(f)$, **SMT_C** 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 **SMT_C** is stronger than **SMT_T**. This implies that a test case that distinguishes an **SMT_C** mutant for statement S , will also distinguish an **SMT_T** 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, \dots, 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--)  
    c = s[i], s[i] = s[j], s[j] = c;
```

F8

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. */  
    s[i] = s[j], s[j] = c, c = s[i];  
for (i = 0, j = strlen(s) - 1; i < j; i++, j--)
```

M18

```
/* Another left rotation generates this mutant. */  
    s[j] = c, c = s[i], s[i] = s[j];
```

M19

When **SSOM** is applied to the **for** statement in the above program, it generates two additional mutants, one by mutating the expression ($i = 0$, $j = \text{strlen}(s) - 1$) to ($j = \text{strlen}(s) - 1$, $i = 0$), 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.

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*.

The semicolon is considered as the ending brace in a compound statement without opening and closing braces.

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--) {
        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--);
        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 $d = e! = c_1 \&\& e! = c_2 \&\& \dots e! = c_n$. 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;
    }
    putc('\n');
case '\r':
    putc('\r');
    break;
}

```

F11

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 may be mutated by the **VSCR** operator. $screen$ itself may be mutated by one of the structure 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.

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

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 $s.c_1.c_2 \dots c_n$ 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;
    int d[10];
}
struct examples, r;
```

F12

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;
}
struct tnode *q;
```

F13

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 $a[e_1][e_2] \dots [e_n]$ with e_i , $1 \leq i \leq n$, 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:

$$\begin{aligned}
& a[e_n] [e_1] \dots [e_{n-2}] [e_{n-1}] \\
& a[e_{n-1}] [e_n] \dots [e_{n-3}] [e_{n-2}] \\
& \quad \vdots \\
& a[e_2] [e_3] \dots [e_n] [e_1]
\end{aligned}$$

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 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.

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`.

```

*i = trap_on_zero_integer(*j) + *k ++
*i = trap_on_positive_integer(*j) + *k ++
*i = trap_on_negative_integer(*j) + *k ++
*i = *j + trap_on_zero_integer(*k++)
*i = *j + trap_on_positive_integer(*k++)
*i = *j + trap_on_negative_integer(*k++)
*i = trap_on_zero_integer(*j + *k++)
*i = trap_on_positive_integer(*j + *k++)
*i = 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 μ Java (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

$$P \xrightarrow{\text{MutOp}} Q$$

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 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 follows:

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 $\text{abs}(e)$.
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.

$$x = \text{abs}(y) + z$$

$$x = y + \text{abs}(z)$$

$$x = \text{abs}(y + z)$$

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 Inheritance

Subclassing can be used in Java to hide methods and variables. A subclass C can declare variable x , hiding its declaration, if any, in $\text{parent}(C)$. Similarly, C can declare a method m hiding its declaration, if any, in $\text{parent}(C)$. 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{
    double dist;
    :
}
class farPlanet extends planet{
    double dist;
    :
}

IHD
class farPlanet extends planet{
    double dist;           ==> class farPlanet extends
                           planet{                      // 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 $\text{parent}(C)$.
IHI	Subclasses	Adds a declaration for variable x to a subclass C if x is declared in the $\text{parent}(C)$.
IOD	Methods	Removes declaration of method m from a subclass C if

		m is declared in $\text{parent}(C)$.
IOP	Methods	Inside a subclass, move a call $\text{super}.M(..)$ 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 $\text{parent}(C)$, 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 $\text{parent}(C)$ given that x is overridden in subclass C .
IPC	super calls	Delete the <code>super</code> keyword from the constructor in a subclass C .

```

class planet{
    String name;
    double dist;
    :
}
class farPlant extends planet{    IHI    => class farPlant extends
                                    planet{
                                    :
}

```

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{
    String name;
    Orbit orbit (...){
        oType=high;
        :
    }
    :
}
IOP
=> }
class farPlant extends planet{
    Orbit orbit (...);
    oType=low; super.orbit()
    :
}

```

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{  class farPlanet extends
    void check(...){...}   planet{
    ...
}

```

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{      class farPlanet extends
    ...                           planet{
        ...
        p=super.name          ISK   ...
        ...
    }

```

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 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 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.

[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;    ...

```

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){...};           OMR void init(int i){
void init (int i, String s){...};  =>   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

A few additional operators model Java language specific faults and common programming mistakes. Eight such operators are listed in [Table 8.13](#).

Some mutation operators are specific to Java. These include the JTD that deletes the “this” keyword and a few others.

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.

<pre>Element hydrogen, hisotope; hydrogen=new Element(); hisotope=hydrogen;</pre>	\xrightarrow{EOA}	<pre>Element hydrogen, hisotope; hydrogen=new Element(); hisotope=hydrogen.clone();</pre>
---	---------------------	---

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 <i>clone()</i> .
EOC	Comparison expressions	Replace == by <i>equals</i> .
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

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.

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.

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 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.

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 <i>END</i> 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

Certainly, many of the C operators that could be used to mutate Java programs are not included in Java. The OEEA 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.

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.

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.

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 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.

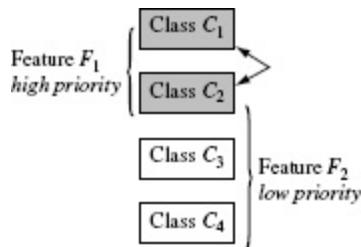


Figure 8.3 Feature F_1 is accorded higher priority than feature F_2 . Hence assign higher priority to classes C_1 and C_2 for mutation. Feature priority is based on criticality, i.e. on the severity of damage in case the program fails when executing a feature.

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?*

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.

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.

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 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.

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

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 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.

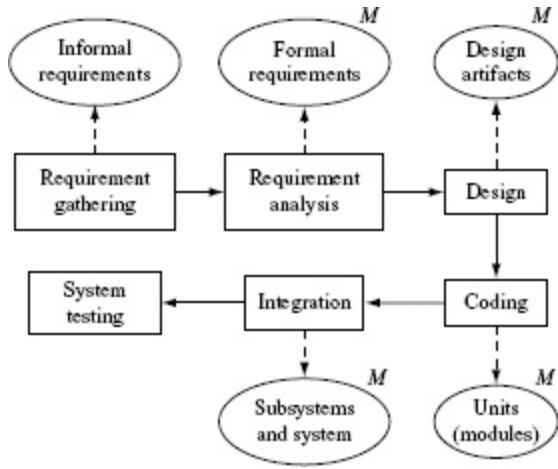


Figure 8.4 Software artifacts generated in a development process. Artifacts that can be subjected to mutation are indicated with an M on the upper right corner. Solid arrows indicate flow of the development process. Dashed arrows indicate production of artifacts in the corresponding stage.

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 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 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.

Some mutation testing tools, such as MuJava, have been integrated with the Eclipse development environment.

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

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.

A variety of mutation testing tools are available. Most are in the public domain and can be obtained by contacting their developers.

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 José 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/
	μ Java	2002	http://www.isse.gmu.edu/~ofut/muJava/
	Lava	2004	http://igor.gold.ac
	Jumble	2012	http://sourceforge.net/projects/jumble/
	Python	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, μ Java, 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:

```
12 if (highCount == 1) danger=moderate;
13 else if (highCount ==2) danger=high;
14 else if (highCount == 3) danger=veryHigh;
15 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 Competent 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 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 μ Java system. Offer arguments in support of, or against, the decision not to include AMC in μ Java.

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 T_EX. Considering a missing initialization as a special case of missing assignment, these faults account for over 15% of all faults found in T_EX. 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.

```

if (q < 1) x=p*q
if((p ≥ -1) ∧ (q > 0)) x=p/q
if((p ≥ 1) ∧ (q > 0)) x=p/q
if((p ≥ 0) ∧ (q > -1)) x=p/q
if((p ≥ 0) ∧ (q > 1)) x=p/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.

```
1  real hamletFunction (x: real) {
2    if (x< 0) x= -x;
3    return(sqr(x)*2)      ← This should be return(sqrt(x) * 2)
4 }
```

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 $T = \{<x = 0>, <x = -1>, <x = 1>\}$. 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 μ Java.

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 conditionl.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.

```
1  bool noSolution=false;
2  int vmmFunction (a, b, c: int){
3    real x;
4    if (a≠ 0){
5      d=b*b-5*a*c;      ← Constant 5 should be 4.
6      if(d<0)
7        x=0;
8      else
9        x=(-b+trunc(sqrt(d))) ;
10    }
11    else
12      x=-c div b;
13    if(a*x*x+b*x+c==0)
14      return(x) ;
15    else{
16      noSolution=true;
17      return(0) ;
18    }
19 }
```

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 $d \leq 0$. Is there any error revealing mutant of `vmmFunction` in $M_W \cup M_O$? (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 μ Java. 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 $0 \leq \text{value} < N$.

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 ?

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.

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 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' .

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.

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.

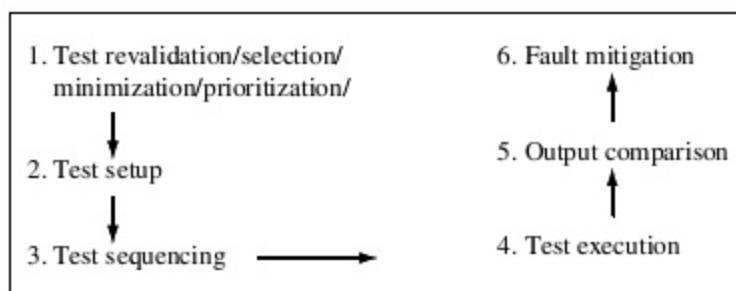


Figure 9.2 A subset of tasks in regression testing.

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

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](#).

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.

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.

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 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.

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.

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 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.

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.

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.

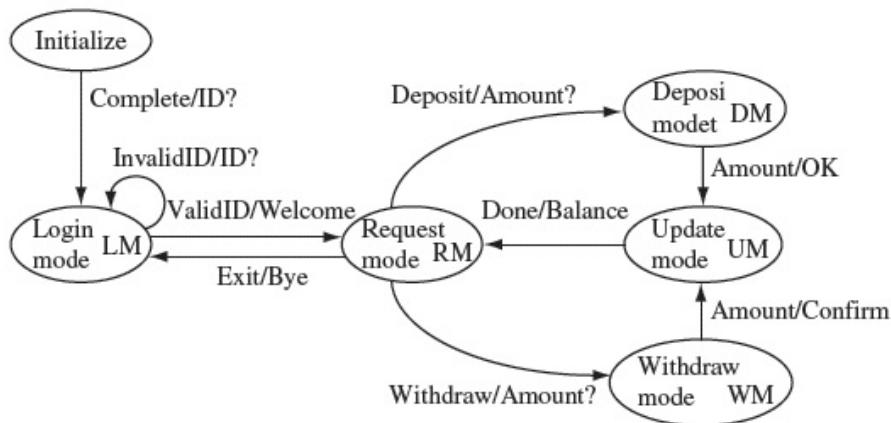


Figure 9.3 State transition in a simplified banking application. Transitions are labeled as X/Y, where X indicates an input and Y the expected output. “Complete” is an internal input indicating that the application moves to the next state upon completion of operations in its current state.

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 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.

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

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.

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.

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.

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 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.

The modified program P must also be tested to ensure the newly implemented code is correct.

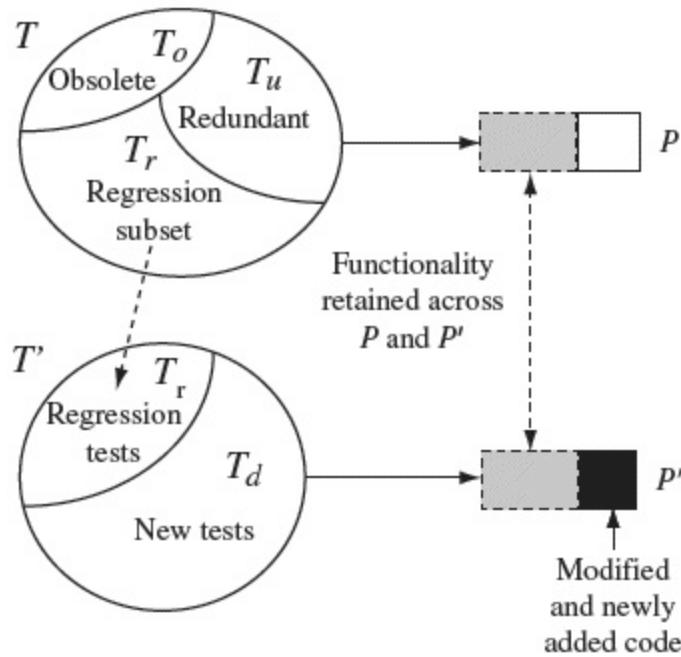


Figure 9.4 Regressing testing as a test selection problem. A subset T_r of set T is selected for retesting the functionality of P that remains unchanged in P' .

Some tests from the previous version of the modified program might become obsolete and will not be included in the regression tests for P' .

In summary, the regression test selection problem (RTS) problem is stated as follows: Find a minimal T_r such that $\forall t \in T_r \text{ and } t' \in T_u \cup T_r, P(t) = P'(t) \Rightarrow P(t') = P'(t')$. 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 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 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 .

Identification of obsolete tests is likely to be a manual activity.

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 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.

Tests that traverse the code that has been modified are good candidates to select for regression testing.

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.

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).

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.

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 truly 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

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' .

An execution trace of a program P is a sequence of program elements executed when P is run against a test.

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 $\text{CFG}(f)$ 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 $\text{trace}(t)$. 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.

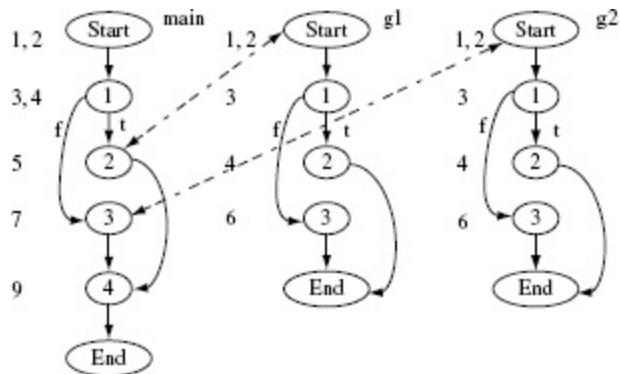


Figure 9.5 Control flow graph for function `main`, and its child functions `g1` and `g2` in [Program 9.1](#). Line numbers corresponding to each basic block, represented as a node, are placed to the left of the corresponding node. Labels `t` and `f` indicate, respectively, true and false values of the 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(){           1  int g1(int a, b){  1  int g2(int a, b){
2  int x, y, p;      2  int a, b;          2  int a, b;
3  input (x, y);    3  if(a+1==b)        3  if(a==(b+1))
4  if (x<y)         4    return(a*a);   4    return(b*b);
5  p=g1(x, y);     5  else             5  else
6  else             6    return(b*b);   6    return(a*a);
7  p=g2(x, y);     7  }               7  }
8  endif
9  output (p);
10 end
11 }
```

Now consider the following test set:

$$T = \left\{ \begin{array}{l} t_1: <x=1, y=3> \\ t_2: <x=2, y=1> \\ t_3: <x=3, y=4> \end{array} \right\}$$

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.

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} .

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.

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, a 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.

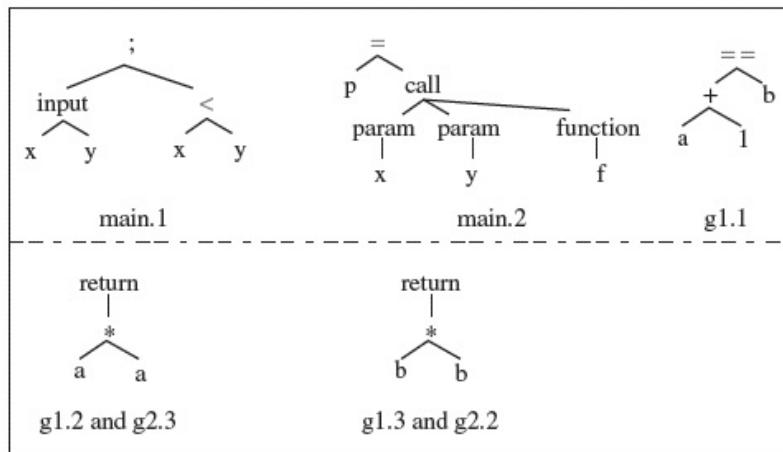


Figure 9.6 Syntax trees for some nodes in the CFGs of functions `main`, `g1`, and `g2` of P shown in [Figure 9.5](#). A semicolon (;) indicates left to right sequencing of two or more statements within a node.

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.

A test t is selected only if the execution traces in P and P' differ, i.e., the traces traverse nodes n_1 and n_2 in the two CFGs that have different syntax trees.

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 .

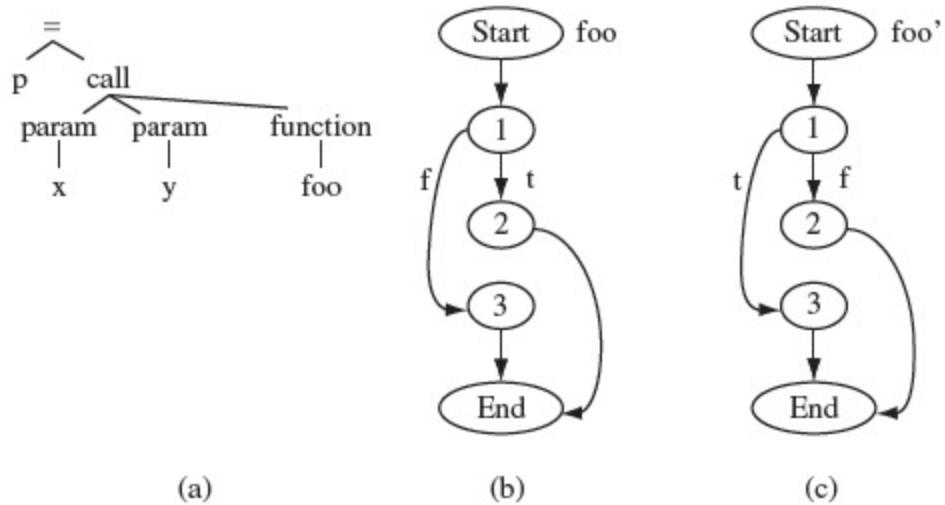


Figure 9.7 Syntax trees with identical function call nodes but different function CFGs. (a) Syntax tree for nodes n and n' , respectively, in G and G' . (b) CFG for `foo` in P . (c) CFG for `foo'` in P' .

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==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: $N = G.\text{main.Start}$ and $N' = G'.\text{main.Start}$. 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 $S = \text{succ}(G.\text{Start}) = \{G.\text{main.1}\}$.

`SelectTests`. Step 4: Let $n=G.\text{main.1}$.

`SelectTests`. Step 4.1: n is unmarked, hence proceed further.

`SelectTests`. Step 4.1.1: $l = \text{label}(G.\text{main.Start}, n) = \epsilon$.

`SelectTests`. Step 4.1.2: $n' = \text{getNode}(\epsilon, G'.\text{main.Start}) = G'.\text{main.1}$.

`SelectTests`. Step 4.1.3: `SelectTests(n, n')`.

`SelectTests`. Step 1: $N = G.\text{main.1}$ and $N' = G'.\text{main.1}$. Mark `G.main.1`.

SelectTests.Step 2 $G.\text{main.1}$ and $G'.\text{main.1}$ are equivalent hence proceed to the next step.

SelectTests.Step 3 $S = \text{succ}(G.\text{main.l}) = \{G.\text{main.2}, G.\text{main.3}\}$.

SelectTests.Step 4: Let $n = G.\text{main.2}$.

SelectTests.Step 4.1: n is unmarked hence proceed further.

SelectTests.Step 4.1.1: $l = \text{label}(G.\text{main.1}, n) = t$.

SelectTests.Step 4.1.2: $n' = \text{getNode}(l, G'.\text{main.1}) = G'.\text{main.2}$.

SelectTests.Step 4.1.3: $\text{SelectTests}(n, n')$.

SelectTests.Step 1: $N = G.\text{main.2}$ and $N' = G'.\text{main.2}$. Mark $G.\text{main.2}$.

As $G.\text{main.2}$ contains a call to g_1 , the equivalence needs to be checked with respect to the CFGs of g_1 and g_2 . N and N' are not equivalent due to the modification in g_1 . Hence $T' = \text{tests}(N) = \text{tests}(G.\text{main.2}) = \{t_1, t_3\}$. 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.

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.

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.

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 $\text{CFG}(f)$ 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 described 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.

1

Output: T' .

Step 1 Repeat the following step for each node $n \in \text{CFG}(f)$.

- 1.1 if $\text{use}(n) \cap declChange_f \neq \emptyset$ or $\text{use}(n) \cap gdeclChange \neq \emptyset$
then $T' = T' \cup \text{test}(n)$.

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 }
```

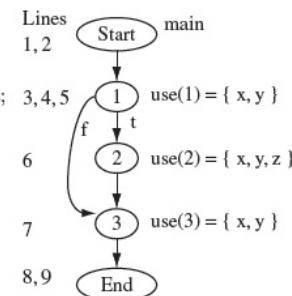


Figure 9.8 Program and its CFG for Example 9.11.

$$T = \left\{ \begin{array}{l} t_1: \langle x=1, y=3 \rangle \\ t_2: \langle x=2, y=1 \rangle \\ t_3: \langle x=3, y=4 \rangle \end{array} \right\}$$

We can easily trace $\text{CFG}(P)$ for each test case to find the test vectors. These are listed below:

<i>test(1):</i>	$\{t_1, t_2, t_3\}$
<i>test(2):</i>	$\{t_1, t_3\}$
<i>test(3):</i>	$\{t_1, t_2, t_3\}$

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 $T = \{t_1, t_3\}$.

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 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 .

Selection of regression tests using the execution trace method may lead to a larger than necessary test suite.

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 }
```

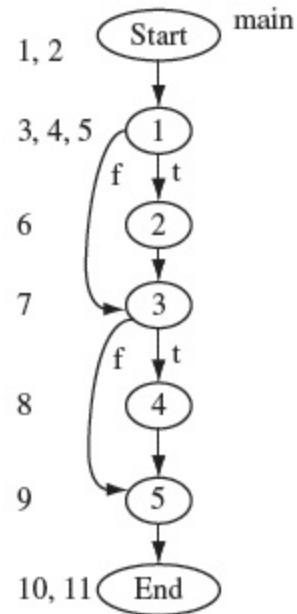


Figure 9.9 A program and its CFG for Example 9.12.

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 $\text{CFG}(P)$. 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 :

$$T = \left\{ \begin{array}{ll} t_1: & \langle p=1, q=3, r=2 \rangle \\ t_2: & \langle p=3, q=1, r=0 \rangle \\ t_3: & \langle p=1, q=3, r=0 \rangle \end{array} \right\}$$

Tests t_1 and t_3 traverse node 2 in $\text{CFG}(P)$ 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

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.

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 .

Example 9.13 Consider P to be Program P9.4 in Figure 9.9 and test t : $\langle p = 1, q = 3, r = 2 \rangle$. 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 : $\langle p = 1, q = 0, r = 0 \rangle$, the dynamic slice contains 3, 4, 5, and 7. The static slice does not change.

9.6.2 Computation of dynamic slices

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.

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.

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.

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 .

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.

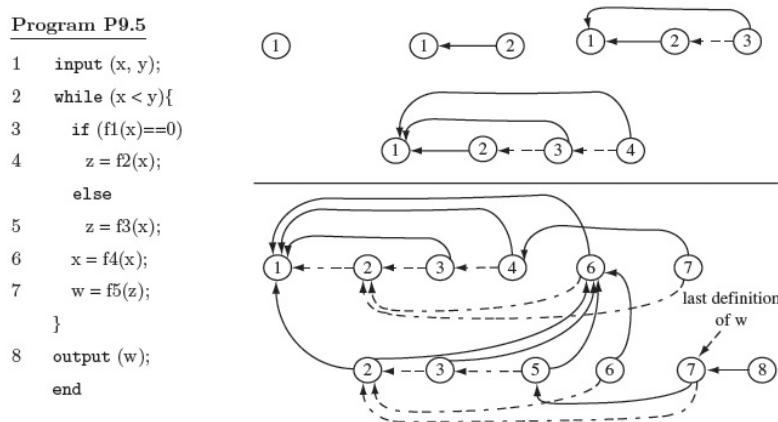


Figure 9.10 A program and its dynamic dependence graph for [Example 9.14](#) and test $< x = 2, y = 4 >$. Function header and declarations omitted for simplicity. Construction process for the first four nodes is shown above the solid line.

The 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](#).

The dynamic program dependence graph is used to obtain dynamic slices.

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

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 $DS(t)$.

Dynamic slices are computed for all or a selected subset of the output variables of a program.

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 undetected.

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.

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:

Potential dependance is used to include test cases that would otherwise be excluded when using the dynamic slicing algorithm mentioned earlier.

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.

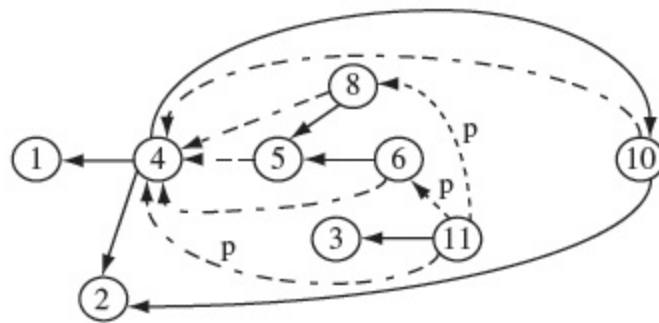


Figure 9.11 Dynamic dependence graph for Program P9.6 obtained from $\text{trace}(t)$, $t: \langle N = 1, x = 1 \rangle$. Unmarked dotted edges indicate control dependence. Edges marked “ p ” indicate potential dependence.

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 .
 - $\text{trace}(t)$ for test t obtained by executing P against t .
 - $\text{DDG}(t)$.

- 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.

*/

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 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 .

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 $PD = \emptyset$.
- 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: $PD = \emptyset$.

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 $nodeSeq = (3^{NV}, 4^{NV}, 5^{NV}, 6^{NV}, 7^{NV}, 8^{NV}, 10^{NV}, 4^{NV}, 11^{NV})$

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. $nodeSeq = (3^{NV}, 4^{NV}, 5^{NV}, 6^V, 8^V, 10^V, 4^V, 11^V)$.

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

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 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.

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

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' .

Modification of a program via the addition or deletion of statements may affect some relevant slices.

- Step 1 Find the set S of statements $s_1, s_2, \dots, s_k, k \geq 0$, that use x . Let n_t , $1 \leq i \leq k$ 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) $n \in RS(t, w, I)$ and (b) $m \in RS(t, w, I)$ 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

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 loses 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.

The size of a dynamic dependence graph is unbounded as it is directly

proportional to that of the execution 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.

A reduced dynamic dependence graph contains at most one occurrence of each node in the program trace.

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](#).

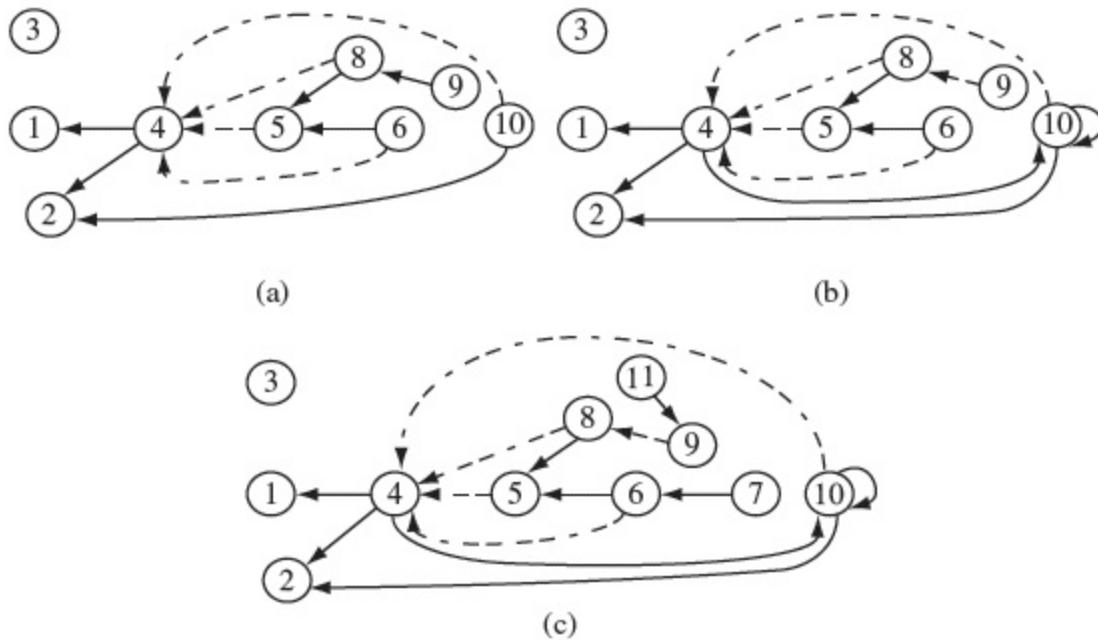


Figure 9.12 Reduced dynamic dependence graph for Program 9.17 obtained from $\text{trace}(t) = \{1, 2, 3, 4^1, 5, 6, 8, 9, 10, | 4^2, 5^2, 6^2, 8^2, 10^2, | 4^3, 5^3, 6^3, 7, 8^3, 10^3, | 4^4, 11\}$.

Intermediate status starting from (a) node 1 until node 10, (b) node 4^2 until node 10^2 , and (c) complete RDDG for $\text{trace}(t)$.

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 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.

For very large programs, one might use coarse-level data and control dependence analysis to reduce the cost of generating dynamic dependence graphs.

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 ($trace(t)$)
$t1$	main, <code>g1</code> , main
$t2$	main, <code>g2</code> , main
$t3$	main, <code>g1</code> , 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 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.

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 }

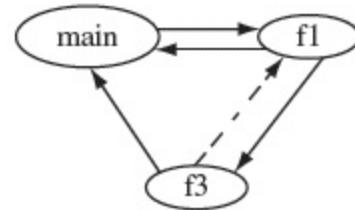


Figure 9.13 Dynamic data dependence graph for Program [P9.7](#) corresponding to the function trace: `main, f 1, f 3, f 1, and main`.

Example 9.24 Program [P9.7](#) consists of `main` and three functions f_1 , f_2 , and f_3 . 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 f_1 due to its use of f_1 to compute the value of z . Function f_1 also has data dependence on `main` due to its use of x . f_3 uses global variable y and hence has data dependence on `main`. Also, f_3 is control dependent on f_1 as it may or may not be called upon entry to f_1 (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

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' .

Test minimization is performed using one of more coverage criteria. These criteria could be black- or white-box.

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 f_2 and f_3 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.

$t_1 : \text{main} : 1, 2, 3$ $f_1 : 1, 3$

$t_2 : \text{main} : 1, 3$ $f_1 : 1, 3$

$t_3 : \text{main} : 1, 3$ $f_1 : 1, 2, 3$

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).

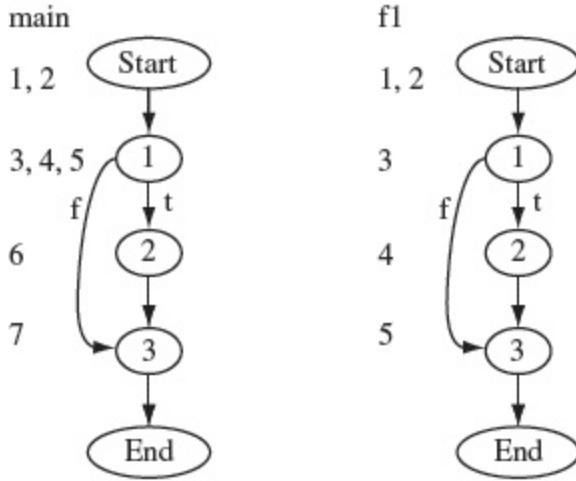


Figure 9.14 Control flow graph of Program P9.7.

9.8.1 The set cover problem

The test minimization problem can 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 .

The test minimization problem can be formulated as a set cover problem.

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:

$$\begin{aligned}
 E &= \{\text{main.1}, \text{main.2}, \text{main.3}, \text{f1.1}, \text{f1.2}, \text{f1.3}\} \\
 TE &= \{\{\text{main.1}, \text{main.2}, \text{main.3}, \text{f1.1}, \text{f1.3}\}, \\
 &\quad \{\text{main.1}, \text{main.3}, \text{f1.1}, \text{f1.3}\}, \\
 &\quad \{\text{main.1}, \text{main.3}, \text{f1.1}, \text{f1.2}, \text{f1.3}\}\}
 \end{aligned}$$

The solution to the test minimization problem is

$$C = \{\{\text{main.1}, \text{main.2}, \text{main.3}, \text{f1.1}, \text{f1.3}\}, \\
 \{\text{main.1}, \text{main.3}, \text{f1.1}, \text{f1.2}, \text{f1.3}\}\}$$

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, 2^{TE} , 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 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.

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 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.

The greedy algorithm might fail to find a minimal test set.

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 $\minCov = \{i_1, i_2, \dots, i_k\}$ 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 $\minCov = \emptyset$. $\text{yetToCover} = m$.

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 $\text{yetToCover} > 0$.

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: $\minCov = \emptyset$. $\text{yetToCover} = 6$.

Step 2: All five tests and six entities are unmarked.

Step 3: As $\text{yetToCover} > 0$, 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. $\text{yetToCover} = 6 - 2 = 4$.

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 , and t_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.

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](#)).

A modified version of the greedy algorithm prioritizes the entities to be covered and then applies the greedy algorithm.

9.9 Test Prioritization

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 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.

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' .

Tests with lower cost are given high priority as compared to tests with higher cost. Of course, one could use multiple criteria to prioritize 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' .

Residual coverage is one criterion used for prioritizing tests.

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 $\langle t_3, t_2, t_1 \rangle$. 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 $PrT = \langle t \rangle$, $X' = X' \setminus \{t\}$. 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 $\text{resCov}(t) \leq \text{resCov}(u)$, 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 . $PrT = \text{append}(PrT, t)$, $X' = X \setminus \{t\}$, and $\text{entitiesCov} = \text{entitiesCov} \setminus \text{cov}(t)$.
- Step 4 Append to PrT any remaining tests in X' . All remaining tests have the same residual coverage which equals $|\text{entitiesCov}|$. 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 ($\text{cov}(t)$)	$ \text{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: $X' = \{t_1, t_2, t_4, t_5\}$. t_5 covers the largest number of functions—13—in X' and hence has the least cost.

Step 2: $PrT = < t_5 >$. $X' = \{t_1, t_2, t_3, t_4\}$.

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 $PrT = \langle t_5, t_3, t_1, t_2, t_4 \rangle$.

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.

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.

There exist several commercial and publicly available tools for regression testing. Few tools actually perform dynamic analysis or test prioritization.

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/ χ Suds 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/ χ Suds 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/ χ Suds	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/ χ Suds 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 f_1 , f_2 , and f_4 , for program P9.5.

$t: < x = 0, y = 3 >, f_1(0) = 0, f_2(0) = 4, f_4(0) = 5$

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      }
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;
9    } // End of first if
10   input (x,y);
11 } // end of while.
12 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. 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.

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;
8    } // End of first if
9  input (x, y);
} // end of while.
9 output (w, z);
end
```

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 loses 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 $E = \{1, 2, \dots, p\}$ where $P = 2^{k+1} - 2$ for some $k > 0$. Let $TE = \{S_1, S_2, \dots, S_k, X, Y\}$, 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) $S_1 \cup S_2 \cup \dots \cup S_k = E$, 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 $E = \{1, 2, 3, 4, 5, 6\}$, $TE = \{S_1, S_2, X, Y\}$, $S_1 = \{1, 2\}, S_2 = \{3, 4, 5, 6\}$, $X = \{1, 3, 4\}$, 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 $X = \{\langle T'_1 \rangle, \langle T'_2 \rangle, \dots, \langle T'_k \rangle\}$, $k > 0$ be a set of sequences of tests such that each sequence $\langle T'_i \rangle$ 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 $\langle T'_i \rangle$ 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 Easy Mock, are introduced. JUnit is a widely used tool for partially automating the unit test process. Easy Mock allows mocking of objects that are not available while testing a unit. The technique of object mocking and its advantages are described using Easy Mock.

10.1 Introduction

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.

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.

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 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.

Unit testing could occur at almost any stage in a software development process. It generally occurs whenever a unit is created.

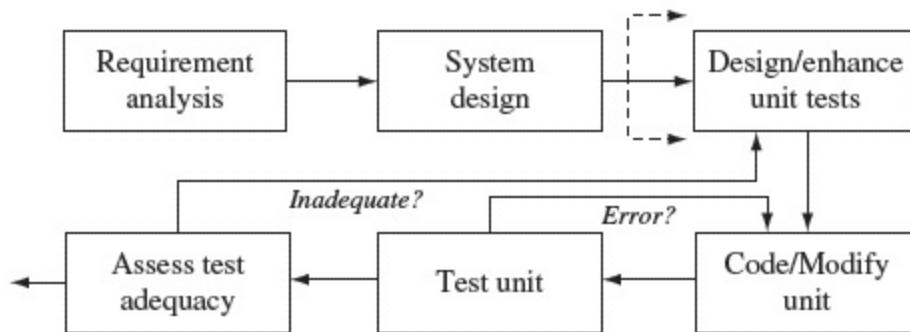


Figure 10.1 One possible context in which unit testing may occur. The process will likely lead to multiple units and their tests though only one unit is shown here. Dotted links indicate continuation of the process for additional units.

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 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.

A unit testing tool, such as JUnit, allows coding of tests and automated execution of a program against these tests.

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).

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 cases can be derived by partitioning the output domain.

Test 1: To test if an exception is raised when the input is invalid.

Input: $\langle a = -1, b = 1, c = 1 \rangle$

Expected response: Exception “Invalid input” must be raised.

Test 2: To test for a right-angled triangle.

Input: $\langle a = 4, b = 4, c = 5.65 \rangle$

Expected response: 1

Test 3: To test for an equilateral triangle.

Input: $\langle a = 1, b = 1, c = 1 \rangle$

Expected response: 2

Test 4: To test for an isosceles triangle.

Input: $\langle a = 1, b = 1, c = 2 \rangle$

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

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](#).

Execution of unit tests can be automated through tools such as JUnit for Java and NUnit for all .Net languages such as C#.

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.

JUnit allows grouping of multiple test into a class.

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`)

```

1 import org.junit.Test;
2 import org.junit.BeforeClass;
3 import static org.junit.Assert.*;
4 import org.junit.Ignore;
5 import junit.framework.*;
6 // Expected Results:
7 // Exception – invalid input . 1 : right angle . 2 : equilateral .
8 // 3 : isosceles . 4 : scalene
9 public class TriangleTest extends TestCase {
10     double maxError; // Max tolerance for right triangle test .
11     double a, b, c; // Lengths of sides .
12     int result; // response from the classifier
13     Triangle t; // Triangle object .
14     @BeforeClass
15     public void setUp () {
16         t = new Triangle ( 1 . 0E-1 );
17     } // End of setUp ( )
18     @Test ( expected=InvalidInputException.class )
19     public void testInvalid () {
20         a = -1 ; b = 1 ; c = 1 ; // Set length of sides
21         result=t . classify ( a , b , c ); // Invoke classifier
22         assertEquals ( 0 , result ); // Check if result is as expected
23     } // End of testInvalid ( )
24     public void testRightAngled () {
25         a = 4 ; b = 4 ; c = 5 . 65 ;
26         result = t . classify ( a , b , c );
27         assertEquals ( 1 , result );
28     } // End of testRightAngled ( )
29     public void testEquilateral () {
30         a = 1 ; b = 1 ; c = 1 ;
31         result = t . classify ( a , b , c );
32         assertEquals ( 2 , result );
33     } // End of testEquilateral ( )
34     public void testIsosceles () {
35         a = 1 ; b = 1 ; c = 2 ;
36         result = t . classify ( a , b , c );
37         assertEquals ( 3 , result );
38     } // End of testIsosceles ( )
39     public void testScalene () {
40         a = 1 ; b = 2 ; c = 3 ;
41         result = t . classify ( a , b , c );
42         assertEquals ( 4 , result );
43     } // End of testScalene ( )
44 } // 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.

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.

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.

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-
TestGeneration/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](#).)

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.

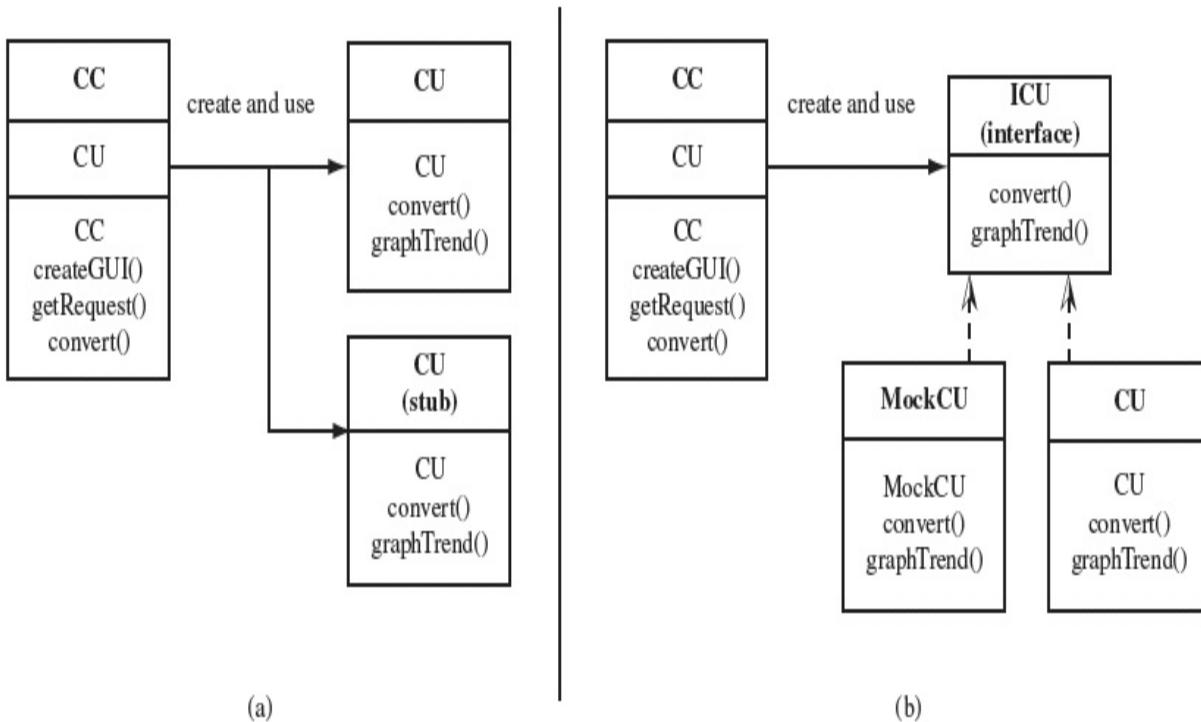


Figure 10.2 Class `cc` is to be tested. `cc` uses class `cu`. (a) Stubbing `cu` when `cc` is not available. (b) Mocking `cu` using the inversion control pattern when `cu` is not available.

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.

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.

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.

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.

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.

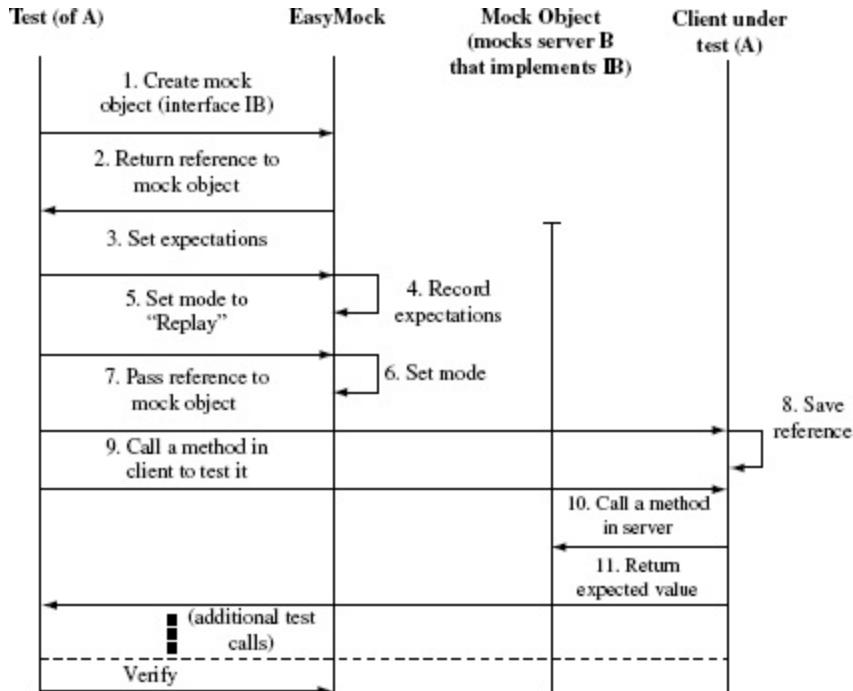


Figure 10.3 Sequence of actions when using the EasyMock tool for unit testing. A sample sequence of actions for one test is shown. A is a client class under test that uses server class B that implements interface IB. The mock object mocks B and allows A to be tested without B.

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 compute
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);
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 :
25              " + result );
26      } // End of save ( )
27  } // 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 class.
10  * Every method starting with the word "test" is called
11  * when running the test with JUnit.
12  */
13 public class TestCompute extends TestCase {
14     private ExpertInterface mockExpert;
15     private Compute calcSave;
16     @Before
17     public void setUp() {
18         mockExpert=EasyMock.createStrictMock(
19             ExpertInterface.class);
20         calcSave = new Compute();
21         calcSave.setExpert(mockExpert);
22     } // End of setUp()
23     // Check if method returns correct value.
24     public void test1() {
25         EasyMock.expect(mockExpert.calc(5)).andReturn(50);
26         EasyMock.replay(mockExpert);
27         assertEquals(50, calcSave.compute(1,5));
28     } // End of test1()
29     // Check number of times called.
30     public void test2() {
31         EasyMock.expect(mockExpert.calc(-5)).andReturn(0).
32             times(2);
33         EasyMock.replay(mockExpert);
34         assertEquals(0, calcSave.compute(1,-5));
35         assertEquals(0, calcSave.compute(1,-5));
36     } // End of test2()
37     // Check method sequencing.
38     public void test3() {
39         EasyMock.expect(mockExpert.calc(5)).andReturn(4);
```

```

38     mockExpert . save (( String ) EasyMock . anyObject ());
39     EasyMock . replay(mockExpert);
40     assertEquals (4, calcSave . compute(2, 5));
41     EasyMock . verify (mockExpert);
42 } // End of test3()
43 // Check invalid parameter.
44 @Test (expected = InvalidCodeException . class)
45 public void test 4 () {
46     EasyMock . expect (mockExpert . calc(5)) . andReturn (4);
47     EasyMock . replay (mockExpert);
48     assertEquals (4, calcSave . compute(3, 5));
49 }
50 }
```

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.

Mock objects could be “nice” or “strict.”

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 `InvalidCodeException` 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
2     RuntimeException {
3     public InvalidCodeException ( String message ) {
4         super ( message );
5     }
}
```

10.6 Tools

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.

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.

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

- [11.1 Introduction](#)
- [11.2 Integration errors](#)
- [11.3 Dependence](#)
- [11.4 OO versus non-OO programs](#)
- [11.5 Integration hierarchy](#)
- [11.6 Finding a near-optimal test order](#)
- [11.7 Test generation](#)
- [11.8 Test assessment](#)
- [11.9 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

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.

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.

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.

A unit (also referred to as a component) could be as simple as a class in Java, or as complex as a database server.

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?

Integration tests focus on the connections across components. Hence, the aim is to detect errors that may lead to incorrect communication among the components.

In the remainder of this chapter, the terms *class*, *component*, and *module* are used synonymously unless differentiated explicitly.

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 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.

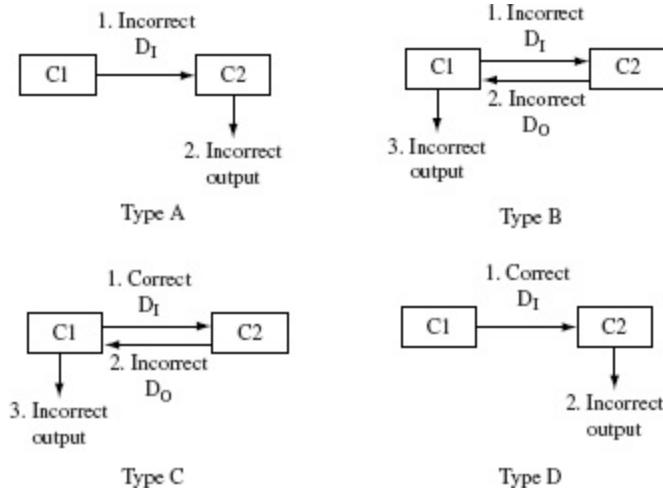


Figure 11.1 Types of integration errors. D_I is a set or a sequence of inputs passed from component C1 to component C2. D_O is a set or a sequence of outputs returned from component C2 to component C1.

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](#).

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*.

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.

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.

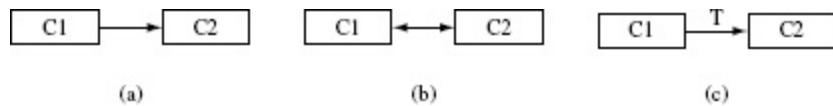


Figure 11.2 (a) Component C1 depends on component C2. (b) Components C1 and C2 depend on each other thus creating a cycle. (c) Component C1 depends on component C2 and the dependency type is T. Here T denotes some form of dependency described in [Section 11.3.1](#).

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.

ORDs can be used to express object relationships in an embedded system consisting of hardware and software.

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.

Obtaining precise relationships among objects might not always be possible, and especially so when the complete code of the system

under test is unavailable.

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 dependence diagram can be drawn and used to decide a suitable integration sequence.

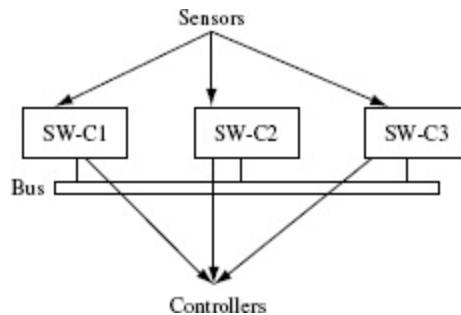


Figure 11.3 A partial chain consisting of software components in the engine control management of an automobile engine control management. Three software components (prefixed as SW-C) are shown. The components communicate via a bus. They also gather data from sensors and send signals to controllers.

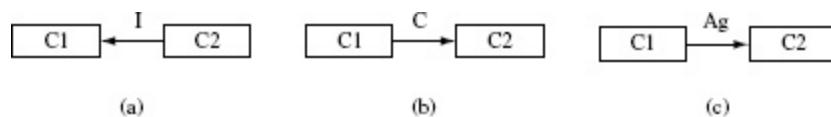


Figure 11.4 (a) Class C2 inherits from class C1. (b) Class C1 aggregates objects of class C2. (c) Class C1 uses data and or methods in class C2.

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:

```
public class C2 extends C1{ // Extension of C1
    ...
}
```

Class relationships could be derived using static analysis of program code. However, relationships thus derived are static and optimistic.

[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 known as *dynamic dependence*. The following example illustrates dynamic dependence.

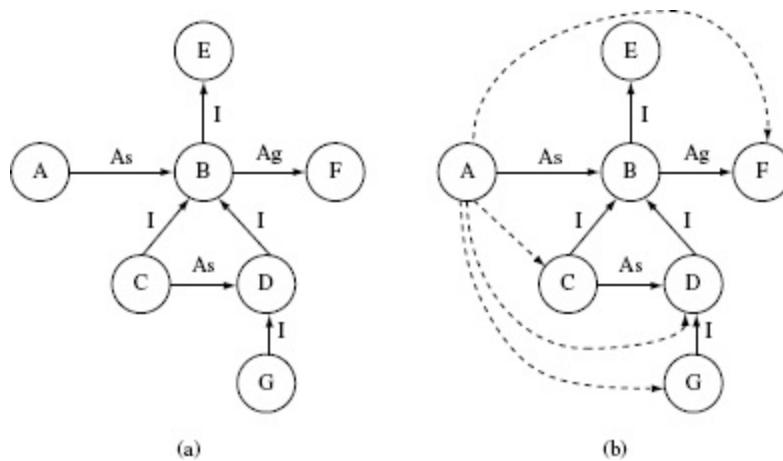


Figure 11.5 (a) Static dependence ORD. (b) Dynamic dependence ORD derived from (a).

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:

$$D = \{(x, y) | \text{when there is an edge in the ORD from class } y \text{ to class } x\}.$$

Let D^+ denote the irreflexive transitive closure of D defined as follows:

$$D^+ = \{(x, z) | (x, y) \text{ and } (y, z) \in D\}.$$

Using D we can define the firewall F_x for class x as follows:

$$F_x = \{z | (x, z) \in D^+\}$$

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:

An ORD can be used to determine the firewall of each class using a “depends” relation.

$$D = \{(B, A), (E, B), (F, B), (B, C), (B, D), (D, C), (D, G)\}.$$

$$D^+ = D \cup \{(B, G), (E, C)(E, D), (E, G), (F, C)(F, D), (F, G)\}.$$

Using D^+ as above, the firewalls for each class in [Figure 11.5\(a\)](#) can be computed as follows:

Class	Firewall	Class	Firewall
A	\emptyset	B	A, C, D, G
C	D	D	G
E	B, C, D, G	F	B, C, D, G
G	\emptyset		

11.3.4 Precise and imprecise relationships

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.

An ORD derived from the code of an application is likely to be more precise than one derived from its design.

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.

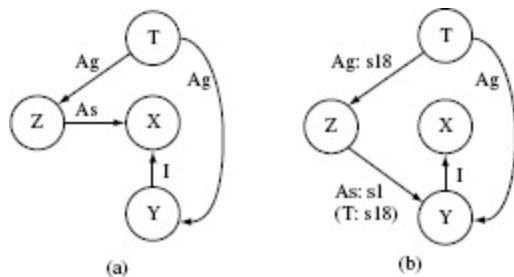


Figure 11.6 ORDs corresponding to Program P11.1. (a) Imprecise ORD. (b) Precise ORD.

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

```
1  public class X{
2      public void print(){
3          System.out.println("This is from X.");
4      }
5  } // End of X
6  public class Y extends X{
7      public void print(){
8          System.out.println("This is from Y.");
9      }
10 } // End of Y
11 public class Z{
12     public Z(X xObject){
13         xObject.print();
14     }
15 } // End of Z
16 public class T{
17     public static void main(String[] args){
18         Z zObject1=new Z(new Y());
19     }
20 } // 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

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.

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

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.

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.

An ORD or a call graph can be used to derive a test order for integration testing.

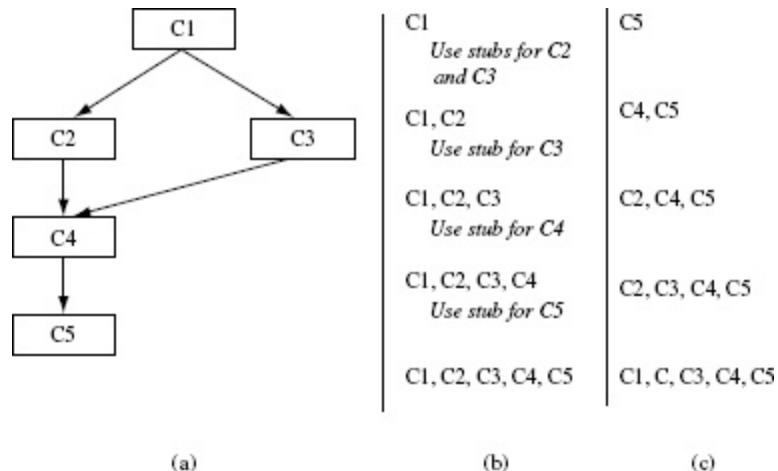


Figure 11.7 (a) A hypothetical set of classes and their relationships. (b) A sample top-down integration sequence. (c) A sample bottom-up integration sequence; drivers are needed for C5, C4, C2, and C3.

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.

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.

The top-down approach integrates components starting from the top and moving down the class hierarchy. Doing so may require many stubs.

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.

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.

The bottom-up approach integrates components starting from the bottom of the class hierarchy and moving up. Doing so may require several drivers.

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, C5} 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.

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.

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.

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 `Graph` 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` and `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](#). 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.

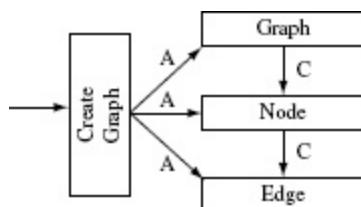


Figure 11.8 Class `Graph` aggregates objects of type `Node` and class `Node` aggregates objects of type `Edge`. Class `CreateGraph` reads node and edge data from a file and creates an object of type `Graph`. “C” indicates the “aggregation” relation which is also known as the “composite,” relation. “A” indicates the association relation.

11.5.2 Comparing integration strategies

Each of the integration strategies discussed above has their pros and cons as examined in the following:

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.

Top-down integration requires the creation of stubs while bottom-up integration requires the creation of drivers.

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 c_1 depends on class c_2 we say that c_1 is a *client* of c_2 . In this case c_2 is considered as a server class. Of course, it is possible that c_1 and c_2 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 c_1 and c_2 is a method m in server class c_2 that is called by c_1 , then retesting of c_1 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 c_1 and c_2 that depend on server class s . If s is not available while testing c_1 or c_2 , 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 s_1 and s_2 as shown in Figure 11.9(c). In either case, when class S is available, c_1 needs to be retested with respect to edge e_1 . Similarly, class c_2 needs to be tested with respect to edge e_2 .

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 class B and the following test sequence: A, E, C, H using $STUB(H, B)$, D, F, B, G .

An ORD might have a cycle. In this situation at least one stub is needed if pure incremental testing is preferred.

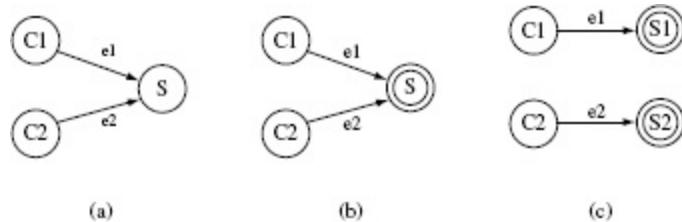


Figure 11.9 (a) Client classes c_1 and c_2 and server class s . (b) Server classes s stubbed. (c) Specific stubs s_1 and s_2 for server class s .

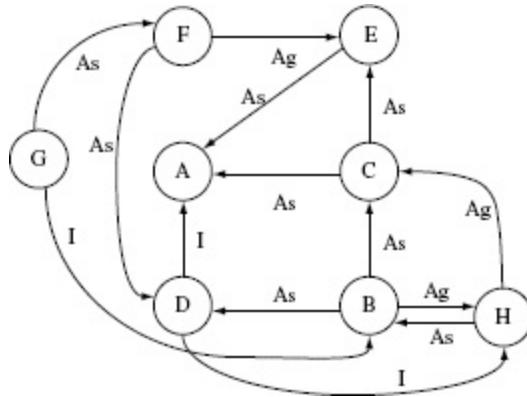


Figure 11.10 A sample ORD with eight classes containing cycles between classes B, H.

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 | <i>Assign major level numbers 1, 2, 3, and so on to each node in G.</i> 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 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 ($\text{maxMajorMinorLevel} < \text{maxLevel}$) then $\text{maxMajorMinorLevel}=\text{maxLevel}$.

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 $\text{maxMajorLevel}=0$. 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 .

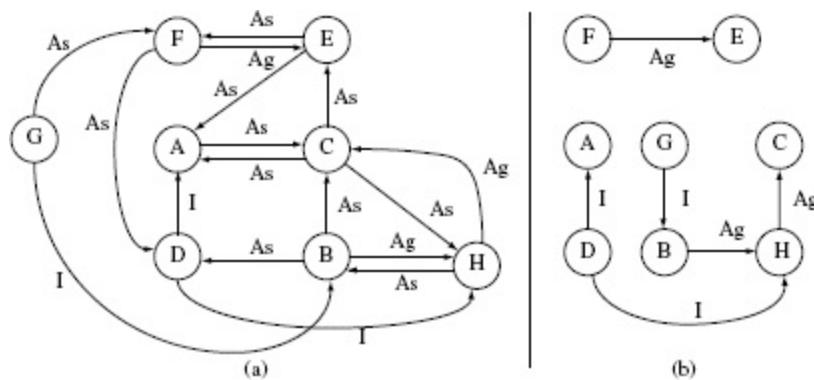


Figure 11.11 (a) An ORD consisting of nodes (classes) labeled A through H. This ORD contains three inheritance edges labeled I, 11 association edges labeled As, and three aggregation edges labeled Ag. (b) ORD in (a) without the association edges. This is used to illustrate the TD method.

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 .

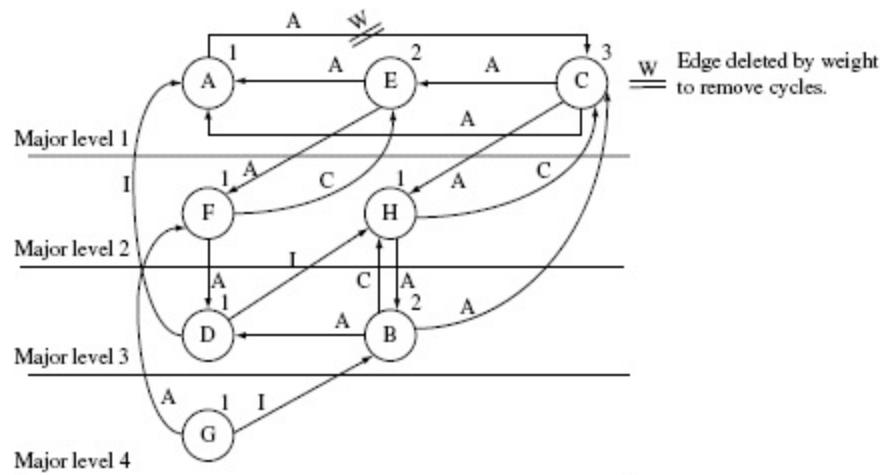


Figure 11.12 Graph G from [Figure 11.11\(a\)](#) with major and minor levels assigned. Nodes at the same major levels are placed just above the corresponding horizontal line. Minor level numbers are placed just above each node. Edge deletion is explained in [Example 11.9](#).

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

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.

Nodes at the same major level are assigned minor level numbers.

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 , $1 \leq i \leq \text{maxMajorLevel}$.

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 n_1 be the source and n_2 the destination node of an edge e . Then the weight of e is $s + d$, where s is the number of edges incident upon n_1 and d is the number of outgoing edges from n_2 .
- Step 6 Let $E_s = \{e_1, e_2, \dots, e_n\}$ be a list of all undeleted edges in G' sorted in the decreasing order of their weights. Thus, the weight of e_1 is greater than or equal to that of e_2 , that of e_2 is greater than or equal to the weight of e_3 , and so on. Delete

from G' the edges in E_s one by one starting with e_1 .
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 $C - A = 1 + 1 = 2$.

`breakCycles`.Step 6: $Es = \{A - C, C - E, E - A, C - A\}$. 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

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.

The test and retest sequence, stubs and retesting edges are determined once the major and minor levels have been assigned.

Procedure for generating test and retest sequence.

Inputs: (a) ORD G with major and minor level numbers assigned.
(b) maxMajorLevel .

(c) $\text{maxMinorLeve}[i]$ for each major level $1 \leq i \leq \text{maxMajorLevel}$.

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 $1 \leq i \leq \text{maxMajorLevel}$.
- 2.1 Execute the following steps until [Step 2.1.9](#) for minor level j , $1 \leq j \leq \text{maxMinorLevel}[i]$.
- 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 $\text{IT} = \{n\}$, where n is the sole node at level(i, j).
- 2.1.2 Update test sequence generated so far. $\text{testSequence} = \text{testSequence} + \text{IT}$.
- 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 $\text{s}_{\text{major}}_n$ be the set of nodes in G at majorlevel greater than i , $i < \text{maxMajorLevel}$, 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 . $\text{SMajor}_n = \emptyset$ for $i = \text{maxMajorLevel}$.

- 2.1.5 For each node $n \in N$, let SMinor_n be the set of nodes in G at minor level greater than j , $j < \text{maxMinorLevel}[i]$, such that there is an association edge from n to each node in SMinor . $\text{SMinor}_n = \emptyset$ for $j = \text{maxMinorLevel}[i]$.
 - 2.1.6 Update the list of stubs generated so far.
 $\text{stubs} = \text{stubs} + \text{SMajor}_n$
 $+ \text{SMinor}_n$ for each $n \in N$.
 - 2.1.7 For each node $n \in N$, let RMajor_n be the set of nodes in G at major level less than i , $i < \text{maxMajorLevel}$, such that there is an edge to n from a node in RMajor . $\text{RetestMajor}_n = \emptyset$ for $i = \text{maxMajorLevel}$. 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 RMinor_n be the set of nodes in G at minor level less than j , $j < \text{maxMinorLevel}[i]$, such that there is an edge to n from a node in RMinor . $\text{RMinor}_n = \emptyset$ for $j = \text{maxMinorLevel}[i]$. Let REMinor be the set of edges to node n from each node in RMinor .
 - 2.1.9 Update the list of retest edges. $\text{retest} = \text{retest} + \text{REMajor}_n + \text{REMinor}_n$ for each $n \in N$.
- Step 3 Return `testSequence`, `stubs`, and `retest`.

End of Procedure `generateTestSequence`

Example 11.10 Let us now apply Procedure `generateTest Sequence` to the ORD in [Figure 11.12](#).

genTestSequence.Step 1: $\text{testSeq} = \emptyset$, $\text{stubs} = \emptyset$, and $\text{retest} = \emptyset$.

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. $\text{IT} = \{A\}$.

genTestSequence.Step 2.1.2: $\text{testSeq} = \{A\}$.

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: SMajor_c .

genTestSequence.Step 2.1.5: $\text{SMinor} = \emptyset$.

genTestSequence.Step 2.1.6: $\text{stubs} = \{C\}$.

genTestSequence.Step 2.1.7: $\text{RMajor}_A = \{\}$. $\text{REMajor}_A = \{\}$.

genTestSequence.Step 2.1.8: $\text{REMajor}_A = \{\}$. $\text{REMajor}_A = \{\}$.

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. $\text{IT} = \{E\}$.

genTestSequence.Step 2.1.2: $\text{testSeq} = \{A, E\}$.

genTestSequence.Step 2.1.3: $N = \{E\}$.

genTestSequence.Step 2.1.4: $\text{SMajor}_E = \{F\}$.

genTestSequence.Step 2.1.5: $\text{SMinor}_E = \emptyset$.

genTestSequence.Step 2.1.6: $\text{stubs} = \{C, F\}$.

genTestSequence.Step 2.1.7: $\text{RMajor}_E = \emptyset$. $\text{REMajor}_E = \emptyset$.

genTestSequence.Step 2.1.8: $\text{RMinor}_E = \emptyset$. $\text{REMinor}_E = \emptyset$.

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. $\text{IT} = \{C\}$.

genTestSequence.Step 2.1.2: $\text{testSeq} = \{A, E, C\}$.

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: $\text{SMajor}_C = \{H\}$.

genTestSequence.Step 2.1.5: $\text{SMinor}_C = \emptyset$.

genTestSequence.Step 2.1.6: $\text{stubs} = \{C, F, H\}$.

genTestSequence.Step 2.1.7: $\text{RMajor}_C = \emptyset$. $\text{REMajor}_C = \emptyset$.

genTestSequence.Step 2.1.8: $\text{RMinor}_C = \emptyset$. $\text{REMinor}_C = \emptyset = \{A-C\}$.

genTestSequence.Step 2.1.9: $\text{retest} = \{A-C\}$. 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 .
- 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.
 - 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. $\text{Stubs} = \text{Stubs} \cup \{m\}$. 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

An edge entering into a node from one of its descendants in a non-trivial strongly connected component is known as a “frond” edge

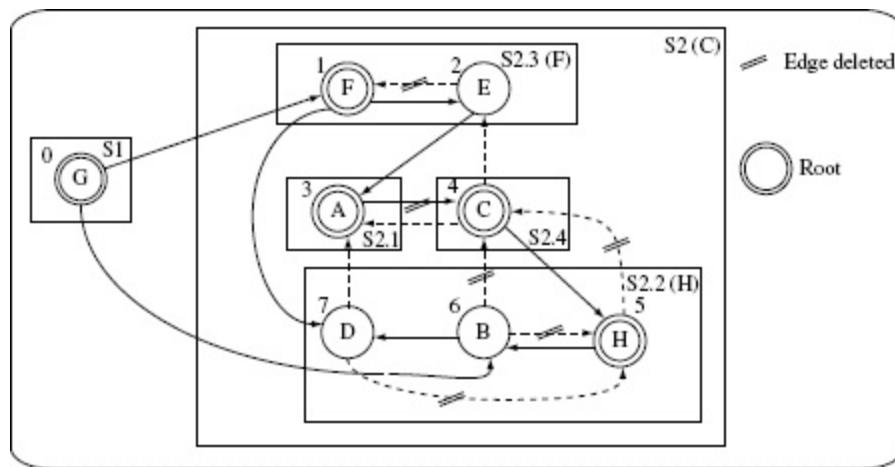


Figure 11.13 Strongly connected components found by applying the TJJM algorithm to the ORD in [Figure 11.11](#). Nodes with two circles are selected roots of an SCC. Each SCC is enclosed inside a rectangle and labeled as $Sx(y)$, where x denotes the SCC number, and when indicated, y denotes the node selected as a root for recursive application to break further cycles. Traversal sequence index is placed against the corresponding node.

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](#). $\text{Stubs} = \text{Stubs} \cup \{C\}$.
- 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 $\cup \{H\}$.
 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.
- 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 $\cup \{F\}$
- 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.

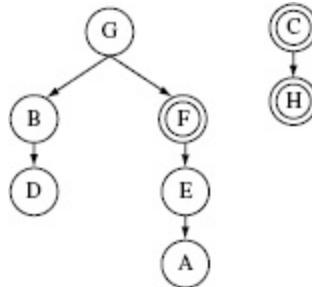


Figure 11.14 DAGs obtained after deleting edge indicated in [Figure 11.13](#).

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 the SCCs in [Step 1. Exercise 11.9](#) asks for applying the algorithm with a different traversal order than used in [Example 11.11](#).

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.

11.6.3 *The BLW method*

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 BLW method creates a test order with the goal of minimizing the number of stubs and specific stubs.

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.

The strategy used to remove cycles from a strongly connected component differentiates the BLW method from the TJM method.

Instead, the BLW method uses the following steps to identify and remove edges from an SCC:

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 TJM method, this step does not use the notion of fronds. Also, in this step the weight of an edge, not a node, is computed.

The BLW method does make use of the edge labels in an ORD.

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 $\text{Stub}(x, Y)$. 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 TJM method which could delete any type of edge.

All steps in the BLW algorithm are the same as shown in Procedure TJM. The BLW algorithm makes use of ideas from the TD and TJM methods. From the TD method it borrows the idea of computing edge weights. From the TJM 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 $\text{Stubs} = \text{Stubs} \cup \{Y\}$. 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 S_1 is shown in [Figure 11.15](#). Thus we have $S = \{S_1\}$. There is only one trivial SCC in this case that consists of node G.

Step 2 Set $s = S_1$.

Step 2.1 The weights of various edges in s are as follows:

$$\begin{array}{llll} \underline{H-B} = (3*3) 9 & C-H = (3*2) 6 & \underline{A-C} = (3*3) 9 \\ B-C = (1*3) 3 & E-A = (2*1) 2 & C-A = (3*1) 3 \\ F-D = (1*2) 2 & B-D = (1*2) 2 & E-F = (2*2) 4 \\ C-E = (3*3) 6 & & \end{array}$$

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. $\text{Stubs} = \text{Stubs} \cup \{B\}$.

Step 2.3 Set $G = s$ and execute procedure BLW.

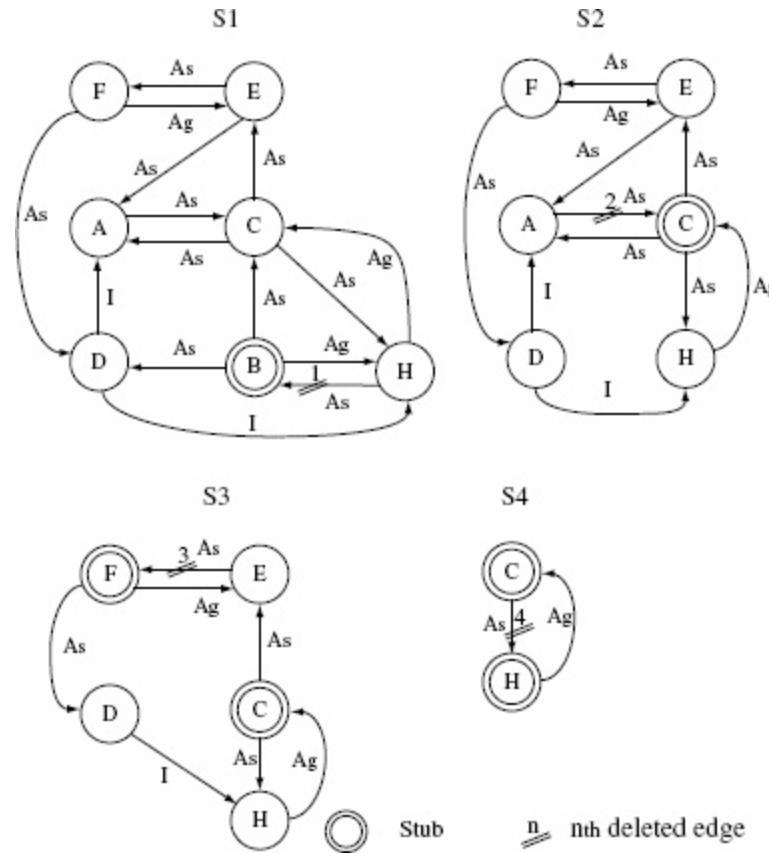


Figure 11.15 SCCs, deleted edges, and stubs obtained by applying the BLW algorithm to the ORD in [Figure 11.13](#). Only the non-trivial SCCs, i.e., those with one or more cycles, are shown.

Step 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:

$$\begin{aligned}
 F-D &= (1*2) 2 & E-F &= (2*2) 4 & C-E &= (2*2) 4 \\
 C-H &= (2*1) 2 & E-A &= (2*1) 2 & C-A &= (2*1) 2 \\
 \underline{A-C} &= (3*3) 9
 \end{aligned}$$

Step 2.2 Edge A-C has the highest weight and is removed from s . Hence, node C is marked as a stub. $\text{Stubs} = \text{Stubs} \cup \{C\}$.

- 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 S_3 in [Figure 11.15](#). The trivial SCC consists of the lone node A.
- Step 2 Set $s = S_3$.
- Step 2.1 The weights of various edges in s are as follows:
- $$\begin{array}{lll} E-F = (2*2) 4 & C-E = (1*1) 1 & C-E = (2*2) 4 \\ \underline{C-H} = (1*1) 1 & F-D = (1*1) 1 & \end{array}$$
- Step 2.2 Edge E–F has the highest weight and is removed from s . Hence, node F is marked as a stub. $\text{Stubs} = \text{Stubs} \cup \{F\}$.
- 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 S_4 in [Figure 11.15](#). The trivial SCCs are {E}, {D}, and {F}.
- Step 2 Set $s = S_4$.
- 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. $\text{Stubs} = \text{Stubs} \cup \{F\}$.
- 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 11.11](#), can now be traversed to obtain the following test order:

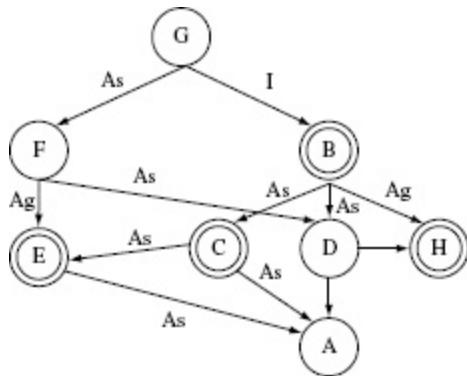


Figure 11.16 DAG obtained by applying procedure BLW to the ORD in [Figure 11.11](#).

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, TJM, and the BLW methods

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.

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.

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		
			TD	Stubs (Specific stubs) 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)

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.

In the experiments the density of the ORD had a significant impact on the number of stubs and specific stubs generated.

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.

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.

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.

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 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.

The TD method appears to be useful when clusters of classes need to be tested together rather than tested in a purely incremental manner.

11.7 Test Generation

11.7.1 *Data variety*

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.

The primary objective of tests generated for integration testing is to test the connections across various components that have been integrated.

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.

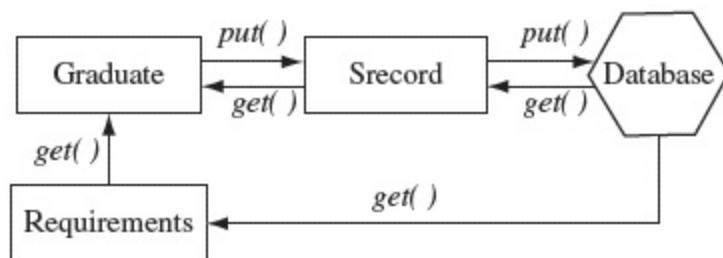


Figure 11.17 A subset of classes in a student record management system. The arrows indicate the direction of data flow. Simply reverse the arrows to obtain the ORD relationships described earlier. *get()* and *put()* are methods to read and write data.

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

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.

Errors that lead to incorrect calls to methods across components might not be revealed during unit testing.

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 $m1()$.

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](#).

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.

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 simple requirements-based adequacy criterion is satisfied when the application was found to behave as expected on all tests.

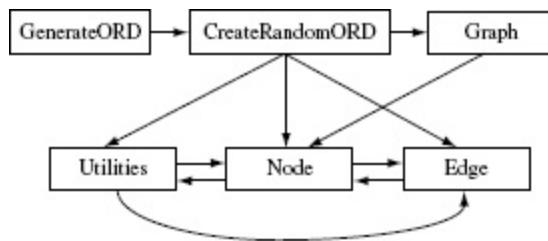


Figure 11.18 Components and their relationships in an application to generate a graph randomly. The generated graph is assumed to be an ORD with only association edges.

Example 11.17 Suppose two components c_1 and c_2 are being tested. As shown in [Figure 11.19](#), c_1 has a method m_1 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.

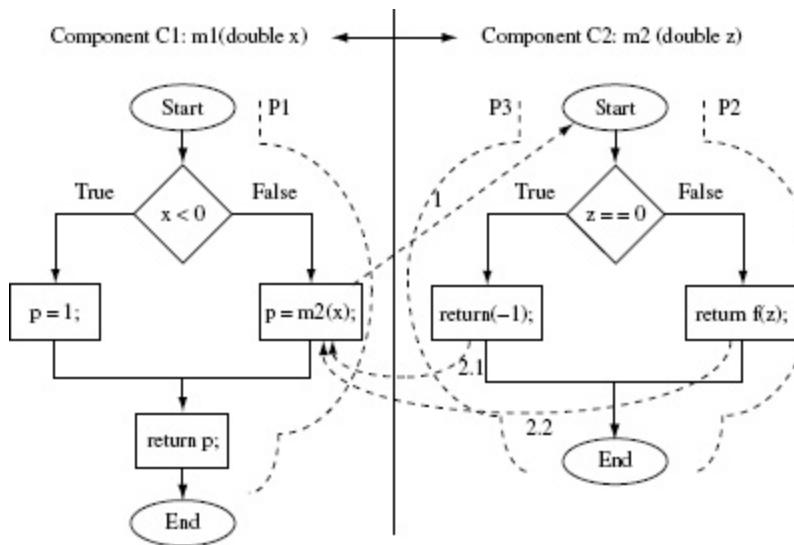


Figure 11.19 Integration of components $C1$ and $C2$. Function f takes a double value as input and returns a value of the expected type.

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 dotted 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.

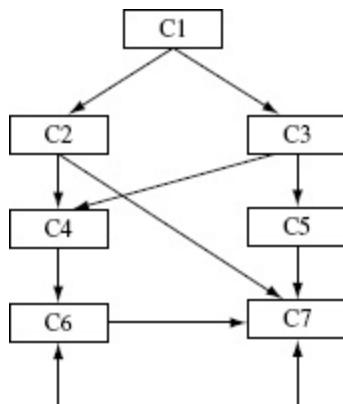


Figure 11.20 A hypothetical component hierarchy for [Exercise 11.1](#). Components C6 and C7 extract data from a database.

- 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.

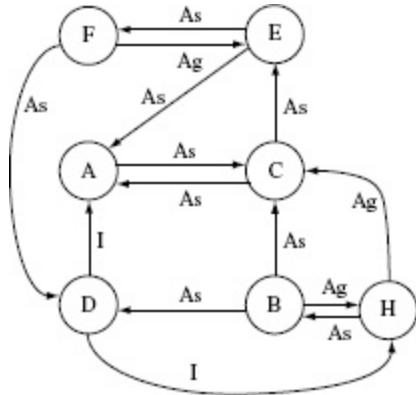


Figure 11.21 A sample ORD with cycles.

11.8 Answer the following with reference to the ORD in Figure 11.22. (a) Find a minimal set of stubs S_1 and the corresponding incremental test sequence. How many specific stubs are required if S_1 is to be used? (b) Find a set of stubs S_2 that minimizes the number of specific stubs. What is the incremental test sequence if S_2 is to be used during testing?

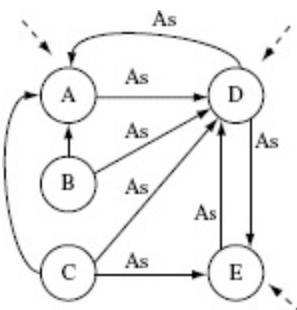


Figure 11.22 A sample ORD with five classes containing cycles between classes A, D and D, E. Dashed arrows indicate potential stubs.

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 TJM and BLW methods.
- 11.21 (a) Apply the TD, TJM, 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 c_1 and c_2 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 c_1 and c_2 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.

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 χ Suds toolset. I continue to believe that χ Suds 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 T_EX and to his staff for sharing with me the earlier versions of T_EX. Hiralal Agrawal for patiently answering questions related to dynamic slicing. Farokh Bastani, Fevzi Belli, Jim Berger, Kai-Yuan Cai, Ram Chillarege, Sid Dalai, 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, José 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 Heddya, 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, Joao 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 AurÓlio Graciotto Silva, Praveen

Ranjan Srivastava, Natalia Silvis, Bryan Tomayko, Rakesh Veera-macheneni, 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, Pilani, 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.

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 present in this eBook at any time, as deemed necessary.

ISBN 9788131794760

ePub ISBN 9789332517660

Head Office: A-8(A), Sector 62, Knowledge Boulevard, 7th Floor, NOIDA 201 309, India.

Registered Office: 11 Community Centre, Panchsheel Park, New Delhi 110 017, India.

FOUNDATIONS OF SOFTWARE TESTING | 2E



ADITYA P. MATHUR

The second edition of *Foundations of Software Testing* is aimed at undergraduate and graduate students as well as practising engineers. It presents sound engineering approaches for test generation, selection, minimization, assessment, enhancement, and sequencing. Using numerous examples, it offers a lucid description of a wide range of simple to complex techniques for a variety of testing-related tasks. It exposes the reader to various commercially available testing tools to facilitate tool selection.

New to this edition

- > Chapters on unit and integration testing
- > Sections on test management, covering topics such as test planning, management and automation
- > Sections on principles of testing, differences between white-box and black-box testing, domain testing and scenario-based testing
- > Summary of various testing tools

The book elucidates the fundamental and advanced testing techniques, replete with varied illustrations.

—B. SATEESH KUMAR, Jawaharlal Nehru Technological University, Hyderabad

It explains all important topics clearly and precisely with enlightening examples throughout the book.

—HARALD A. STIEBER, Nuremberg Institute of Technology, Germany

Illustrated with good examples, the book offers a perfect blend of theoretical concepts and practical aspects.

—RAHUL RISHI, Maharishi Dayanand University, Haryana

It is an ideal book that covers diverse software testing principles and techniques by means of good examples and comprehensive explanations.

—SARFARAZ MASOOD, Jamia Millia Islamia, New Delhi

This new edition of *Foundations of Software Testing* has improved in terms of focus on both theory and practice and could serve both as a good textbook for students and also a reference for practitioners.

—VAHID GAROUSI, University of Calgary, Canada

Online Student Resources

» http://wps.pearsoned.co.in/mathur_fost_2

*see inside cover for details

For Instructors: Lecture Slides

For Students: Testing Tools, Additional Reading Material



Online Instructor resources available at
www.pearsoned.co.in/adityapmathur



Cover photo: Shutterstock

www.pearsoned.co.in