Software & Security Testing

Course Notes for SWEN90006

The School of Computing and Software Systems

The University of Melbourne

Aims, Assessment and Course Outline

Software Processes and Management versus Engineering Methods

Software engineering is about exercising *control* over the software that you are developing. To be specific, it means exercising control over the functional and non-functional aspects of the software, as well as the timelines, budgets, and resources needed to build a software system. The focus of subjects on project management is on understanding processes and the principles used for engineering software and for running I.T. projects. For example, the key phases in any software engineering project are the requirements, design, implementation, testing and maintenance phases. In short:

> Software engineering project management is about exercising control over the *process* of building a software system.

In contrast *SWEN90006* is about exercising control over the *product* and its quality. But quality is a vague concept that cannot be controlled or measured directly. The common approach in software engineering is to break the concept of quality down into a number of *attributes* that are measurable, or at least can be evaluated to some degree. Exercising control over the measurable attributes in turn allows us to exercise control over the quality of the software system, the product. In short:

> *SWEN90006* is about understanding how to exercise control over the attributes of the product.

System Attributes, Techniques and Tools

> *What are these system attributes, techniques and tools that we are talking about?*

The following is a short, but certainly incomplete, list of the sorts of things that you will need to keep a close watch over in order to achieve high quality in your software.

Functional Correctness, Completeness and Consistency Perhaps first in most people's minds are these *three C's* of correctness, completeness and consistency.

Correctness can mean different things to different people. For example, do we judge correctness with respect to the requirements specification or the design models; or is correctness to be judged against the purpose and intent of the clients? If your software is *not* correct then you may have violated the contract for your system. At best, users will avoid your software and find some-one else's program that they can rely on.

The *completeness* of a piece of software usually refers to the extent to which the system implements functions to solve the original problem. It can refer to whether or not your software system implements all of the requirements, or whether or not your software system implements all of the functions needed by the stakeholders.

The *consistency* of a piece of software usually refers to whether or not the various functions implement in the system deliver consistent results with each other or in successive runs of the system. For example, we would

always want a spreadsheet like Excel to deliver the same results every time we applied one of its statistical functions.

Completeness and consistency have similar issues in interpretation as correctness. When judging the completeness and consistency of a program do we judge against the requirements specification, the design models or the purpose of the program.

Security is an attribute that is increasingly important as people and organisations put more and more of their lives and livelihoods online. Security is different to correctness in that it is not an implicit requirement that we assess against.

When software engineers measuring the correctness of a system, they take requirements and verify that the system implements those using different methods; e.g., testing, reviews. However, many security breaches are caused by the things that the engineers do *not* think about during development, rather than incorrect implementations of what they did think about. Backdoors, overflows, etc., are not violations of requirements per se, but are vulnerabilities that do not relate to specific requirements — at least, in many cases. Thus, the methods and techniques used to assess and measure security are not just functional testing and review (although these form an important part of it), but must go further[1].

Reliability is an attribute often confused with correctness. Reliability, while similar to correctness, is nevertheless different.

Correctness is a simple attribute of a software system — your software system is either correct or it is not. Reliability, on the other hand, is the probability that a system will not *fail* for a given period of time in a given environment. Correctness is assessed on a pass/fail basis by using testing methods aimed at finding program failures[2]. Reliability is typically assessed using *random testing* and *Monte Carlo methods* to estimate probabilities of failure [3].

Performance is an attribute that is often confused with *program efficiency*. The two concepts are certainly related but different in their approach and methods. Efficiency is usually used to describe a program that computes its results in minimal time and using minimal resources while performance deals in methods that can be used to meet specific time or resource targets.

Performance is measured by observing a collection of system variables such as the average time to complete a task, the throughput of a data in a system, the hard real-time responses to events, memory usage and resource usage in general.

Performance is important for all systems because poor performance can affect usability and consequently your ability to market and sell products. Further, for hard real-time systems performance is critical as systems will often have to respond to inputs within a fixed period of time.

Usability is a system property that relates to the ease of use of the system and the ability of users to achieve their tasks quickly, accurately and, when necessary, in a safe way.

Building a *usable* system requires a specific focus on usability at all stages of development and this often includes significant user models as well as conducting the proper monitoring processes through-out the development of the system.

Other subjects in the curriculum provide a more comprehensive introduction to the design of interactive graphical systems and consequently usability is not covered in this subject.

There are other attributes of a system that are important as well: *extendability*, *dependability*, and *survivability* to name but a few.

Engineering a software system means being confident that you have implemented *all* of the functions required by the stakeholders of the system *and* being able to estimate, design, measure and control the critical attributes for your program. *Ideally, you should be able to specify the target values for each of the attributes, design them into your software system, and then control your system development in such a way that you reach the specified targets.*

[1] See Chapter 8
[2] See Chapters 1—6.
[3] See Chapter 7

In practice, it is not so easy because programs are large and complex, and attributes such as usability and safety have no directly observable quantities that can be used to measure them.

A concrete example is needed. Consider a *reliability growth model*[4] and the number of failures experienced per unit time during the testing phases of development. When we experience a failure (usually caused by a program fault) then we would normally fix the fault and retest. We would expect the number of failures experienced per unit time to fall as the process of finding and fixing faults continues. The graph in Figure 1 shows how this process can be measured and what the graph of *failures per unit time* (*R*) against the time spent in testing and debugging might look like. However, in practice we may well find instead a pattern such as the one in Figure 2. A peak in

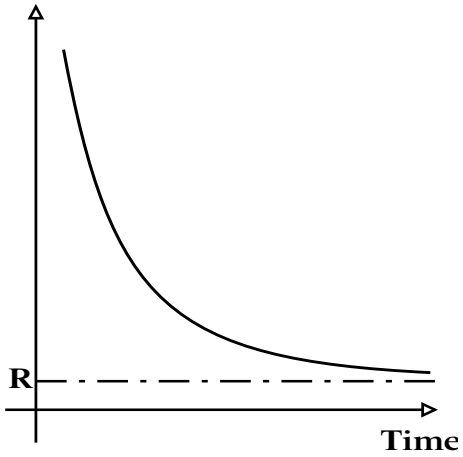


Figure 1: The expected reliability growth according to Musa simple execution time model.

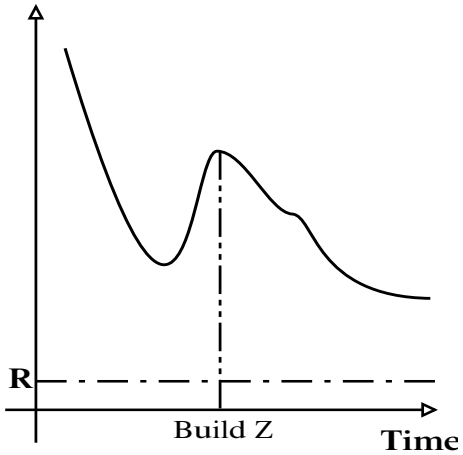


Figure 2: The actual reliability as experienced during testing.

failures per unit time is experienced around *build Z* and we are now in jeopardy of requiring much more testing than originally planned in order to get back to our original reliability targets.

Now the project management must decide if:

- the peak is due to normal project circumstances, for example a change in requirements or the integration of a new module, or
- the peak is due to a problem with the development process, for example, design faults in the system or a misinterpretation of requirements or one of the sub-teams not adhering to proper standards.

[4] See Chapter 7

The point here is not what the problem is or whose fault it is, but that by measuring reliability and using the reliability growth model, problems in the development can be uncovered, and corrected. *Now, control is being exercised.*

Subject Overview

In this subject we will explore the following topics. The aim is not to try and be all things to all people but to give a fairly detailed treatment of some specialised engineering methods.

Systems Engineering is where we will begin. The aim is to set the context for much of what follows from a systems viewpoint. Software is typically part of a larger system that involves different operating systems and hardware, networks and other software.

The point here is that you need to open up your vision, especially in the engineering process, to think about the larger system.

There are some critical sections from your previous experience to recall here:

(i) Software Requirements specifications;

(ii) Software Architectures and the various forms of coupling such as data coupling and control coupling; and most importantly

(iii) Software Quality.

Please go back and revise them now.

Quality in our final product is what we are seeking. By choosing the right engineering methods to use in our overall development processes we can improve the quality of our software – or at least that is the assumption that we will make in this subject.

The aim here is to establish the relationship between our engineering methods, system requirements and system quality. In short, it is to understand how and why engineering methods can help us to meet the goal of engineering a system with quality.

Software Testing is an engineering method for verifying correctness, used in reliability and for establishing performance. Testing methods are thus central to SWEN90006 and, in our view, central to experimenting with systems.

The aim is to explore systematic methods for testing software, selecting test inputs to maximise testing coverage, and the implications of testing for the correctness and completeness of a piece of software.

Software Reliability Engineering follows on from testing. We will explore reliability models for software, reliability measures, reliability and testing, and how to use reliability engineering principles to achieve more confidence in the correctness of software.

Security Testing leaves behind the idea of functional correctness, complete, and related topics, and aims to determine: how secure is our system? Here, we will explore common security vulnerabilities, how to test effectively to detect these, and advanced symbolic methods for detecting these.

Generic Skills

The subject is a technical subject. The aim is to explore the various approaches to building software with specific attributes into a system.

The subject will also aim to develop a number of *generic skills*.

- We encourage *independent research* in order to develop your ability to learn independently, assess what you learn, and apply what you learn.

- We will be conducting the projects in teams which will, of course, require *teamwork*. While not the primary goal of the project you are expected to be disciplined in your approach to the projects by requiring that rigorous processes be followed and that you meet certain milestones along the way.
- You will also be developing experience at empirical software engineering, and the interpretation and assessment of quantitative and qualitative data.
- Finally, and perhaps most importantly, you will be encouraged to develop critical analysis skills that will complement and round out what you have learned so far in your degree, and help you to think more deeply about all phases of software development.

Contact Details

Tim Miller
Room: 6.09, Doug McDonell Building
Email: tmiller@unimelb.edu.au

Toby Murray
Room: 8.17, Doug McDonell Building
Email: toby.murray@unimelb.edu.au

Expectations on Students

The subject has formal teaching through lectures and tutorials/workshops. There are also consultation times for you.

Lectures and Lecture Notes

These subject notes are the main reference for the subject. They contain notes on all of the major topics contained in the subject, as well as a number of appendices that are included for interest and to round fill out the the material in the notes. *Material in the appendices will not be examined*

You are expected to read the subject notes.

In addition to the subject notes, additional material will be handed out in lectures. Lectures are aimed at providing you with another view of the material in the subject notes. Lectures will primarily consist of problem-solving exercises and discussions, so it is expected that you will gain a deeper understanding of concepts and material from lectures than from the lecture notes.

You are expected to attend lectures.

Material from other subjects such as Algorithms and Data Structures, Object-oriented Software Development, and Software Processes and Management is assumed in the subject notes and in the lecture slides. Reference books are reserved for overnight loan at the Engineering library and for certain topics papers may also be handed out in the lectures.

The staff of SWEN90006 will expect you to be familiar with the material in the subject notes prior to tutorials and workshops and certainly for assignments and projects.

Tutorials/Workshops

The aim of the tutorials is to create a bridge between theory and practice. The tutorial questions are similar in standard to those in the assignments and the final exam. They are also aimed at getting you to think about the ways in which the theory can be applied to practical evaluation problems.

The staff of SWEN90006 will expect you to actively engage in tutorials by conducting tutorial exercises and in engaging in the discussions.

Consultations

Consultations are for you to ask questions about assessment work or to seek a deeper understanding of the subject material. You are encouraged to come to consultations whenever you are having difficulty with the subject matter or the assessment work.

Assessment

This subject, like many software engineering subjects, is a blend of theory and practice. The practical assessment consists of three assignments and a team project.

Assessment	Topic	Type	Percentage
Assignment 1	Software testing	Individual	20%
Peerwise	Choice by students	Individual	5%
Assignment 2	Security testing	Team	25%
Exam	All topics	Individual	50%

Assignments

There are four assignments in the subject. Two of these are worth 2.5% each, and the other two are worth 20% and 25% respectively.

The two smaller assignments will require you to submit a multiple-choice question with a sample answer into the *PeerWise* repository: `http://peerwise.cs.auckland.ac.nz/`. The aim of these assignments is to encourage deeper thinking about what you have learned in the subject. It also provides the class with a repository of sample questions during revision, and allows class members to peer review and comment on each others questions and answers.

To get the full 2.5%, you will be required to demonstrate understanding of the subject content, to provide a quality question, and to provide a helpful answer.

The two larger assignments will require you to apply theory learned in lectures to the analysis and testing of programs. There are limits to testing and analysis of programs — so each of the assignments will ask some technical questions as well as some questions aimed at getting you to explore these limitations.

What we are looking for in assignment work is understanding as well as practical application. You assignments will only be graded above 80% if you demonstrate both understanding and technical ability adequately.

In short — we give marks if you show thinking

The aim of the group project project is for you to build up your skills in security analysis, your evaluation skills, and your research skills. The group project is worth 25%.

The project will be group-based. You are expected to work as part of a team, and will be assessed as a team. This is to ensure that everyone has 'skin in the game' — which simply means that there is an incentive to work as a team instead of as a group pursuing individual interests.

Examination

The examination is a 2 hour exam based on the lecture, tutorial and assignment material. The tutorial questions are written in the same style and of similar difficulty to the exam questions. Being able to answer the assignment questions, tutorial questions, and project work means that you will be well prepared for the exam.

Hurdles

Important note: This subject, like many in the school, has a hurdle on both exam and coursework. This means that to pass the subject, you will need to pass both the coursework and the exam.

Assessment Component	**Percentage of Total**	**Hurdle**
Coursework assignments	50%	25%
Final Examination	50%	25%

Contents

List of Figures

List of Tables

Chapter 1

An Introduction to Software Testing

Testing is an integral part of the process of *assuring*[1] that a system, program or program module is suitable for the purpose for which it was built. In most textbooks on software engineering, *testing* is described as part of the process of *validation* and *verification*. The definitions of validation and verification as described in *IEEE-Std. 610.12-1990* are briefly described below.

Validation is the process of evaluating a system or component during or at the end of the development process to determine if it satisfies the requirements of the system. In other words, validation aims to answer the question:

Are we building the correct system?

Verification is the process of evaluating a system or component at the end of a phase to determine if it satisfies the conditions imposed at the start of that phase. In other words, verification aims to answer the question:

Are we building the system correctly?

In this subject, testing will be used for much more than just validating and verifying that a program is fit for purpose. We will use testing, and especially random testing methods, to *measure* the attributes of programs. **Note** that not all attributes of a program can be quantified.

Some attributes, like reliability, performance measures, and availability are straightforward to measure. Others, such as usability or safety must be estimated using the engineer's judgement using data gathered from other sources. For example, we may estimate the safety of a computer control system for automated braking from the reliability of the braking computers and their software.

Before looking at testing techniques we will need to understand something of the *semantics* of programs as well as the *processes* by which software systems are developed. First we look at the process of building up systems from their components in Section 1.1. The relationship between integration (or creating system *builds*) and testing is an important one. The order in which modules are integrated and tested can make the process of finding faults easier if a good strategy is chosen and may result in fewer latent faults[2].

It is necessary to understand the *semantics* of programs so that, as a tester, we can be more confident that we have explored the program thoroughly. We discuss programs briefly in Section 1.2.

[1] Recall from *Software Processes and Management* that the word *assurance* means having confidence that a program or document or some other artifact is fit for purpose. Later in these notes we will try and translate the informal and vague notion of *confidence* into a measure of probability or strict bounds.

[2] Informally, *latent faults* are faults that remain undiscovered during development until after the system has been released and a specific state of the system or specific external circumstances trigger the fault.

1.1 Testing and Integration

The process of *developing* and *debugging* your programs, *building* your programs, *validating* and *verifying* your programs and *testing* your programs are inter-related and depend upon each other.

In general terms, a *software system* is a collection of programs. You may be developing a single (often referred to as a *monolithic*) program, or you may need to develop a *suite* of programs that must work together to achieve a common goal. We will use the word "system" to mean either a single program or a collection of programs.

The development of a system typically follows a process that involves requirements, design, implementation and testing phases. The process also typically uses various methods for validating and verifying the documents, plans, programs and other artifacts that are developed in each phase. This is where the various life-cycles and techniques from the Software Processes and Management subject are used.

Integrating the System

In the process of designing a system, you will have decomposed your system into modules, packages or classes. If the system is complicated then you may have decomposed each module into sub-modules and so on.

Integrating your system refers to putting the modules, packages or classes together to create subsystems, and eventually the system itself. Generally speaking, integration and testing go together. Good integration strategies can give help you to minimise testing effort required. Poor integration strategies generally cause more work, and lead to lower quality software.

Normally, a system integration is organised into a series of *builds*. Each build combines and tests one subset of the modules, packages or objects of the system.

The aim for most integration strategies is to divide the modules into subsets based on their *dependencies*. Typical build strategies then try and integrate all those modules with no dependencies first, the modules that depend on these second, and so on. Here are some examples of integration strategies.

The Big-bang Integration Strategy

The *Big-Bang* method involves coding all of the modules in a sub-system, or indeed the whole system, separately and then combining them all at once. Each module is typically unit tested first, before integrating it into the system. The modules can be implemented in any order so programmers can work in parallel.

Once integrated, the completely integrated system is tested. The problem is that if the integrated systems fails a test then it is often difficult to determine which module or interface caused the failure. There is also a huge load on resources when modules are combined (machine demand and personnel).

The other problem with a Big-Bang integration strategy is that the number of input combinations becomes extremely large — testing all of the possible input combinations is often impossible and consequently we there may be latent faults in the integrated sub-system

A Top-Down Integration Strategy

The *top-down* method involves implementing and integrating the modules on which no other modules depend, first. The modules on which the these *top-level* modules depend are implemented and integrated next and so on until the whole system is complete.

The advantages over a Big-Bang approach are that the machine demand is spread throughout integration phase. Also, if a module fails a test then it is easier to isolate the faults leading to those failures. Because testing is done incrementally, is is more straightforward to explore the input for program faults.

The top-down integration strategy requires *stubs* to be written. A stub is an implementation used to stand in

for some other program. A stub may simulate the behaviour of an existing implementation, or be a temporary substitute for a yet-to-be-developed implementation.

The Bottom-Up Integration Strategy

The *bottom-up* method is essentially the opposite of the top-down. Lowest-level modules are implemented first, then modules at the next level up are implemented forming subsystems and so on until the whole system is complete and integrated.

This is a common method for integration of object-oriented programs, starting with the testing and integration of base classes, and then integrating the classes that depend on the base classes and so on.

The bottom-up integration strategy requires *drivers* to be written. A driver is a piece of code use to supply input data to another piece of code. Typically, the piece modules being tested need to have input data supplied to them via their interfaces. The driver program typically calls the modules under test supplying the input data for the tests as it does so.

A Threads-Based Integration Strategy

The *threads-based* integration method attempts to gain all of the advantages of the top-down and bottom-up methods while avoiding their weaknesses. The idea is to is to select a minimal set of modules that perform a program function or program capability, called a **thread**, and then integrate and test this set of modules using either the top-down or bottom-up strategy.

Ideally, a thread should include some I/O and some processing where the modules are selected from all levels of the module hierarchy. Once a thread is tested and built other modules can be added to start building up a complete the system.

Different Types of Testing

There are various types of testing that systems typically undergo. Each type of testing is aimed at detecting different kinds of failures and making different kinds of measurements. Figure 1.1 is a representative diagram that shows how each type of testing corresponds to a stage in the development[3].

Unit Testing

The first task in unit testing is to work out exactly what a unit is for the purpose of testing. In most cases, a unit is a module: a collection of procedures or functions that manipulate a shared state. In these cases, the module is tested as a whole, because changes in one procedure can effect others. In other cases, a unit is a single procedure or function. In these cases, the only inputs and outputs are parameters and return values — there is no state.

Units, if they are functions or classes, are tested for correctness, completeness and consistency. Unit testing measures the attributes of units. They can be tested for performance but almost never for reliability; units are far too small in size to have the statistical properties required for reliability modelling.

If each unit is thoroughly tested before integration there will be far fewer faults to track down later when they are harder to find.

[3]The diagram in Figure 1.1 is called the *V Model*. The V Model is not (necessarily) a process model. Instead, it is used to demonstrate that you design your system tests against requirements specifications, integration tests against high level design specifications and unit tests against detailed design specifications.

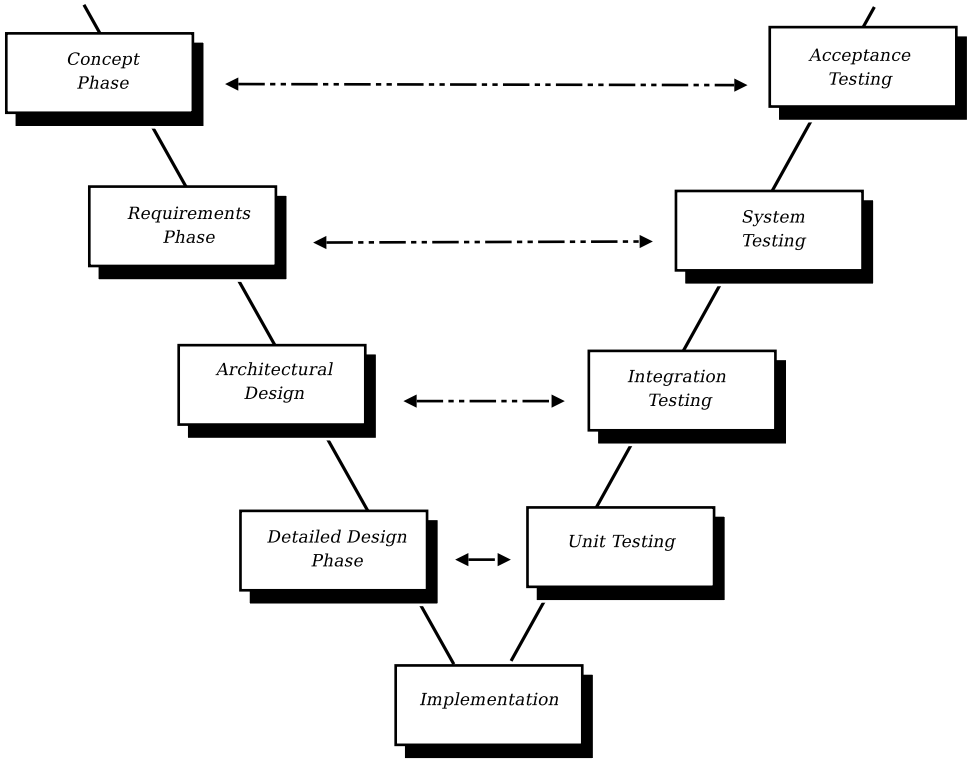


Figure 1.1: A simplified view of the relation between testing levels and development artifacts.

Integration Testing

Integration testing tests collections of modules working together. Which collection of modules are integrated and tested depends on the integration strategy. The aim is to test the interfaces between modules to confirm that the modules interface together properly. Integration testing also aims to test that subsystems meet their requirements.

Integration tests are derived from high level component designs and requirements specifications.

System Testing

System testing test the entire system against the requirements, use cases or scenarios from requirements specification and design goals. Sometimes system testing is broken down into three groups of related testing activities:

Definition 1

(i) **System Functional Test**: tests the entire system against the functional requirements and other external sources that determine requirements such as the user manual.

(ii) **System Performance Test**: test the non-functional requirements of the system for example, the load that the system can handle, response times and can test usability (although this latter testing is more like a survey than what we would call an actual *test*).

(iii) **System Acceptance Test**: is a set of tests that the software must pass before it is accepted by the clients.

Regression testing

Programs undergo changes but the changes do not always effect the entire program. Rather certain functions or modules are changed to reflect new requirements, system evolution or even just bug fixes. History shows that, after making a change to a system, testing only the part of the system that has changed is not enough. A change in one part of a system can effect other parts of the system in ways that are difficult to predict. Therefore, after any change, the entire test suite of a system must be run again. This is called *regression testing*.

Regression testing is one of the key reasons for wanting to be able to execute test suites automatically. To test an entire system by hand is costly even for the smallest of systems, and infeasible for many others.

1.2 Programs

To be a good tester, it is important to have a sound conceptual understanding of the *semantics of programs*. When it comes to software there are a number of possibilities, each of which has its own set of complications.

Firstly, lets consider the simple function given in Figures 1.2, written in an imperative programming language like C.

```
void squeeze(char s[], int c)
{
    int i,j;
    for (i = j = 0; s[i] != '\0'; i++) {
        if (s[i] != c) {
            s[j++] = s[i];
        }
    }
    s[j] = '\0';
}
```

Figure 1.2: The squeeze function from Kernighan and Ritchie implemented in C.

The function squeeze removes any occurrence of the character denoted by the variable c from the array s, and squeezes the resulting array together. The semantics of C are such that integers and characters are somewhat interchangeable, in that an element of the type int can be treated as a char, and vice-versa.

If the squeeze function were written in Haskell then its type would be

$$\mathsf{squeeze} \ : \ \mathsf{string} \times \mathsf{char} \rightarrow \mathsf{string}.$$

The input type is string $\times$ char and the output type is string. The output type is implicit because the parameter to squeeze is a call by reference parameter.

The set of values in the input type is called the *input domain* and the set of values in the output type is called the *output domain*. For functions like the squeeze function, the input domain is the set of values in the input type, and the output domain is the set of values in the output type.

The fibonacci function, shown in Figure 1.3, takes an integer N and returns the sum of the first N Fibonacci numbers[4].

Again, if the fibonacci function were written in Haskell its type would be

$$\mathsf{fibonacci} \ : \ \mathsf{unsigned\ int} \rightarrow \mathsf{unsigned\ int}$$

and so the input domain for the fibonacci function is the set of values of type unsigned int. Likewise, the output domain for the fibonacci function is also the set of values of type unsigned int.

[4]The Fibonacci numbers is a sequence of numbers given by $1\ 1\ 2\ 3\ 5\ \ldots\ f_n\ \ldots$ where $\mathsf{f_n} = \mathsf{f_{n-1}} + \mathsf{f_{n-2}}$.

```
unsigned int fibonacci(unsigned int N) {
  unsigned int a = 0, b = 1;
  unsigned int sum = 0;
  while (N > 0) {
    unsigned int t = b;
    b = b + a;
    a = t;
    N = N - 1;
    sum = sum + a;
  }
  return sum;
}
```

Figure 1.3: A function to sum the first N Fibonacci numbers implemented in C.

Word Size	unsigned int **Set of Values**	int **Set of Values**
8 Bit Integers	$0 \ldots 255$	$-127 \ldots 128$
16 Bit Integers	$0 \ldots 65,535$	$-32,767 \ldots 32,768$
32 Bit Integers	$0 \ldots 4,294,967,295$	$-2,147,483,647 \ldots 2,147,483,648$

Table 1.1: The set of values for different word sizes.

The *input domain* to a program is the set of values that are accepted by the program as inputs. For example, if we look at the parameters for the squeeze function they define the set of all pairs (s, c) where s has type array of char and c has type int.

Not all elements of an input domain are relevant to the specification. For example, consider a program that divides a integer by another. If the denominator is 0, then the behaviour is undefined, because a number cannot be divided by 0.

Note The input domain can vary on different machines. For example, the input domain to the fibonacci function is the set of values of type unsigned int but this can certainly vary on different machines. Table 1.1 shows how the set of values for parameters of type unsigned int and int respectively differ for different machine word sizes.

The output domain for the fibonacci function is identical. The design specification of the fibonacci function should specify the range of integers that are *legal* inputs to the function.

Determining the input and output domains of a program, or function is not as easy as it looks, but it is a skill that is vital for effectively selecting test cases. That is why we will spend a good deal of time on input/output domain analysis.

Notice also that in the squeeze and fibonacci functions, for any input to the function (an element in the input domain) there exists a unique output (element output domain) computed by the function.

The function in question is *deterministic*. A function is deterministic if for every input to the function there is a unique output — the output is completely determined by the input[5]. In this case it is easier to test the program because if we choose an input there is only one output to check.

BUT, we do not always have deterministic programs. If a program can return one of a number of possible outputs for any given input then it is *non-deterministic*. Concurrent and distributed programs are often non-deterministic. Non-deterministic programs are harder to check because for a given input there is a set of outputs to check.

Programs may *terminate* or *not terminate*. The squeeze and fibonacci functions both terminate. In the case of fibonacci an output is returned to the calling program and so to test fibonacci we can simply execute it and examine the returned value. The function squeeze returns a void value so even if it terminates we must still examine the array that was passed as input, because its value may change.

[5]Recall that the property that for all inputs there exists a unique output is the defining property of a mathematical function.

On the other hand a classic example of a non-terminating program is a control loop for an interactive program or an embedded system. Control loops effectively execute until the system is shutdown. In the same way as the squeeze function and the fibonacci function a non-terminating program may:

(i) generate observable outputs; or

(ii) not generate any observable outputs at all.

In the former case we can test the program or function by testing that the sequence of values that it produces is what we expect. In the second case we need to examine the internal state somehow.

1.3 Testing Your Programs

Testing, at least in the context of these notes, means *executing* a program in order to measure its attributes. Measuring a program's attributes means that we want to work out if the program *fails* in some way, work out its response time or through-put for certain data sets, its mean time to failure (MTTF), or the speed and accuracy with which users complete their designated tasks.

Our point of view is closest to that of IEEE-Std. 610.12-1990, but there are some different points of view. For comparison we mention these now.

- Establishing confidence that a program does what it is supposed to do (W. Hetzel, *Program Test Methods*, Prentice-Hall)
- The process of executing a program with the intent of finding errors (G.J. Myers, *The Art of Software Testing*, John-Wiley)
- The process of analysing a software item to detect the difference between existing and required conditions (that is, bugs) and to evaluate the features of the software item (IEEE Standard for Software Test Documentation, IEEE Std 829-1983)
- The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component (IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990)

The viewpoint in these notes is deliberately chosen to be the broadest of the definitions above, that is, the last item in the list above. The theme for this subject is that testing is not just about detecting the presence of faults, but that testing is about *evaluating attributes of system and its components*. This means software testing methods are used to *evaluate* and *assure* that a program meets all of its requirements, both functional and non-functional.

To be more specific, software testing means executing a program or its components in order to assure:

A. The correctness of software with respect to requirements or intent;

B. The performance of software under various conditions;

C. The robustness of software, that is, its ability to handle erroneous inputs and unanticipated conditions;

D. The security of software, that is, the absence of vulnerabilities and its robustness against different kinds of attack;

E. The usability of software under various conditions;

F. The reliability, availability, survivability or other dependability measures of software; or

G. Installability and other facets of a software release.

Remark

(i) Testing is based on *executing* a program, or its components. Therefore, testing can only be done when parts (at least) of the system have been built.

(ii) Some authors include V&V activities such as audits, technical reviews, inspections and walk-throughs as part of testing. We do not take this view of testing and consider reviews, inspections, walk-throughs and audits as part of the V&V process but not part of testing.

1.3.1 The Purpose of Testing

> "*Program testing can be used to show the presence of bugs, but never to show their absence!*" — Edsger W. Dijkstra[6]

This quote states that the purpose of testing it to demonstrate that their is a discrepancy between an implementation and its specification, and that it cannot be used to show that the implementation is correct. The aim of most testing methods is to systematically and actively find these discrepancies in the program.

Testing and debugging are NOT the same activity. *Debugging* is the activity of: (1) determining the exact nature and location of a suspected fault within the program; and (2) fixing that fault. Usually, debugging begins with some indication of the existence of an fault. The purpose of debugging is to locate fault and fix them.

Therefore, we say that the aim of testing is to demonstrate that there is faults in a program, while the aim of debugging is to locate the cause of these faults, and remove or repair them.

The aim of *program proving* (aka *formal program verification*) is to show that the program does not contain any faults. The problem with program proving is that most programmers and quality assurance people are not equipped with the necessary skills to prove programs correct.

1.3.2 The Language of Failures, Faults, and Errors

To begin, we need to understand the language of *failures*, *faults* and *errors*.

Definition 2

(i) **Fault:** A fault is an incorrect step, process, or data definition in a computer program. In systems it can be more complicated and may be the result of an incorrect design step or a problem in manufacture.

(ii) **Failure:** A failure occurs when there is a deviation of the observed behaviour of a program, or a system, from its specification. A failure can also occur if the observed behaviour of a system, or program, deviates from its intended behaviour which may not be captured in any specification .

(iii) **Error:** An incorrect internal state that is the result of some fault. An error may not result in a failure – it is possible that an internal state is incorrect but that it does not affect the output.

Failures and errors are the result of faults – a fault in a program can trigger a failure and/or an error under the right circumstances. In normal language, software faults are usually referred to as "bugs", but the term "bug" is ambiguous and can mean to faults, failures, or errors; as such, as will avoid this term.

Consider the following simple program specification: for any integer [7] n, $\mathsf{square}(\mathsf{n}) = \mathsf{n} * \mathsf{n}$; and an (incorrect) implementation of this specification in Figure 1.4.

Executing square(3) results in 6 – a *failure* – because our specification demands that the computed answer should be 9. The *fault* leading to failure occurs in the statement return x*2 and the first *error* occurs when the expression x*2 is calculated.

However, executing square(2) would not have resulted in a failure even though the program still contains the same fault. This is because the behaviour of the **square** function of under the input 2 behaves correctly. This is called

[6] In *Notes on Structured Programming*.

[7] We will not worry about the range just yet.

```
int square(int x)
{
    return x*2;
}
```

Figure 1.4: A faulty squaring function written C.

coincidental correctness. If 2 was chosen as the test input, then by *coincidence*, this happens to exhibit the correct behaviour, even though the function is implemented incorrectly.

The point here is there are some inputs that reveal faults and some that do not. While the above example is trivial, any non-trivial piece of software will display coincidental correctness for many test inputs.

In testing we can only ever detect failures. Our ability to find and remove *faults* in testing is closely tied to our ability to detect failures. The discussion above leads naturally to the following three steps when testing and debugging software components.

(1) Detect system failures by choosing test inputs carefully;

(2) Determine the faults leading to the failures detected;

(3) Repair and remove the faults leading to the failures detected; and

(4) Test the system or software component again.

This process is itself error-prone. We must not only guard against errors that can be made at steps (2) and (3) but also note that new faults can be introduced at step (3). It is important to realise here that steps (2) and (3) must be subject to the same quality assurance activities that one may use to develop code.

1.3.3 Test Activities

Ultimately, testing comes down to the selecting and executing *test cases*. A test case for a specific component consists of three essential pieces of information:

(i) a set of test inputs, or if the program under test is non-terminating, a set of sequences of test inputs;

(ii) the expected results when the inputs are executed; and

(iii) the execution conditions or execution environment in which the inputs are to be executed.

A collection of test cases is a *test suite*.

As part of the process of testing, there are several steps that need to be performed. These steps remain the same from unit testing to system testing.

Test Case Selection

Given the above definition of a test case, there are two generic steps to test case selection. Firstly, one must select the test inputs. This is typically performed using a *test input select technique*, which aims to achieve some form of *coverage criterion*. Test input selection is covered in Chapters 2–6 of these notes.

Secondly, one must provide the expected behaviour of every test input that is generated. This is referred to as the *oracle* problem. In many cases, this oracle can be derived in a straightforward manner from the requirements of the program being tested. For example, a test case that assesses performance of a system may be related to a specific requirement about performance in the requirements specification of that system.

Despite the amount of research activity on software testing, the oracle problem remains a difficult problem, and it is difficult to automate oracles or to assess their quality.

Like any good software engineering process, test case selection is typically performed at a high level to produce abstract test case, and these are then refined into executable test cases.

Test Execution

Once executable test cases are prepared, the next step is to execute the test inputs on the program-under-test, and record the actual behaviour of the software. For example, record the output produced by a functional test input, or measure the time taken to execute a performance test input.

Test execution is one step of the testing process that is generally able to be automated. This not only saves the tester time, but allows for regression testing to be performed at minimal case.

Test Evaluation

Compare the actual behaviour of the program under the test input with the expected behaviour of the program under that test input, and determine whether the actual behaviour satisfies the requirements. For example, in the case of performance testing, determine whether the time taken to run a test is less than the required threshold.

As with test execution, test evaluation can generally be automated.

Test Reporting

The final step is the report the outcome of the testing. This report may be returned to developers so they can fix the faults that relate to the failures, or it may be to a manager to inform them that this stage of the testing process is complete.

Again, certain aspects of test reporting can be automated, depending on the requirements of the report.

1.3.4 Test Planning

Testing is part of quality assurance for a software system, and as such, testing must be *planned* like any other activity of software engineering. A *test plan* allows review of the adequacy and feasibility of the testing process, and review of the quality of the test cases, as well as providing support for maintenance. As a minimum requirement, a test plan should be written for every artifact being tested at every level, and should contain at least the following information:

(i) *Purpose*: identifies what artifact is being tested, and for what purpose; for example, for functional correctness;

(ii) *Assumptions*: any assumptions made about the program being tested;

(iii) *Strategy*: the strategy for test case selection;

(iv) *Supporting artifacts*: a specification of any of the supporting artifacts, such as test stubs or drivers; and

(v) *Test Cases*: an description of the abstract test cases, and how they were derived.

Other information can be included in a test plan, such as the estimate of the amount of resources required to perform the testing.

1.3.5 Testability

The *testability* of software can have a large impact on the amount of testing that is performed, as well as the amount of time that must be spent on the testing process to achieve certain test requirements, such as achieving a coverage criterion.

Testability is composed of two measures: *controllability* and *observability*.

Definition 3 *Controllability* and *Observability*

(i) The *controllability* of a software artifact is the degree to which a tester can provide test inputs to the software.

(ii) The *observability* of a software artifact is the degree to which a tester can observe the behaviour of a software artifact, such as its outputs and its effect on its environment.

For example, the squeeze from Figure 1.2 highly controllable and observable. It is a function whose inputs are restricted to parameters, and whose output is restricted to a return value. This can be controlled and observed using another piece of software.

Software with a user interface, on the other hand, is generally more difficult to control and observe. Test automation software exists to *record* and *playback* test cases for graphical user interfaces, however, the first run of the tests must be performed manually, and expected outputs observed manually. In addition, the record and playback is often unreliable due to the low observability and controllability.

Embedded software is generally less controllable and observable than software with user interfaces. A piece of embedded software that receives inputs from sensors and produces outputs to actuators is likely to be difficult to monitor in such an environment — typically much more difficult than via other software or via a keyboard and screen. While the embedded software may be able to be extracted from its environment and tested as a stand-alone component, testing software in its production environment is still necessary.

Controllability and observability are properties that are difficult to measure, and are aspects that must be considered during the design of software — in other words, software designers must consider *designing for testability*.

1.4 Input Domains

To perform an analysis of the inputs to a program you will need to work out the sets of values making up the input domain. There are essentially two sources where you can get this information:

A. the software requirements and design specifications; and

B. the external variables of the program you are testing.

In the case of white box testing, where we have the program available to us, the input domain can be constructed from the following sources.

A. Inputs passed in as parameters;

B. Inputs entered by the user via the program interface;

C. Inputs that are read in from files;

D. Inputs that are constants and precomputed values;

E. Aspects of the global system state including:

(i) Variables and data structures shared between programs or components;

(ii) Operating system variables and data structures, for example, the state of the scheduler or the process stack;

(iii) The state of files in the file system;

(iv) Saved data from interrupts and interrupt handlers.

In general, the inputs to a program or a function are stored in *program variables*. A program variable may be:

- A variable declared in a program as in the *C* declarations

```
int base;
char s[];
```

- Resulting from a read statement or similar interaction with the environment, for example,

```
scanf("%d\n", x);
```

Variables that are inputs to a function under test can be:

- *atomic data* such as integers and floating point numbers;
- *structured data* such as linked lists, files or trees;
- a reference or a value parameter as in the declaration of a function; or
- constants declared in an enclosing scope of the function under test, for example:

```
#define PI 3.14159

double circumference(double radius)
{
    return 2*PI*radius;
}
```

We all try running our programs on a few test values to make sure that we have not missed anything obvious, but if that is all that we do then we have simply not covered the full input domain with test cases. Systematic testing aims to cover the full input domain with test cases so that we have assurance – confidence in the result – that we have not missed testing a relevant subset of inputs.

1.5 Black-Box and White-Box Test Case Selection Strategies

If we do have a clear requirements or design specification then we choose test cases using the specification. Strategies for choosing test cases from a program or function's specification are referred to as *specification-based* test case selection strategies. Both of the following are specification-based testing strategies.

Black-box Testing where test cases are derived from the functional specification of the system; and

White-box Testing where test cases are derived from the internal design specifications or actual code for the program (sometimes referred to as *glass-box*).

Black-box test case selection can be done without any reference to the program design or the program code. Black-Box test cases test only the functionality and features of the program but not any of its internal operations.

A. The real advantage of black-box test case selection is that it can be done *before* the implementation of a program. This means that black-box test cases can help in getting the design and coding correct with respect to the specification.

   Black-box testing methods are good at testing for missing functions and program behaviour that deviates from the specification. They are ideal for evaluating products that you intend to use in a system such as COTS[8] products and third party software (including open source software).

B. The main disadvantage of black-box testing is that black-box test cases cannot detect additional functions or features that have been added to the code. This is especially important for systems that are safety critical (additional code may interfere with the safety of the system) or need to be secure (additional code may be used to break security).

White-box test cases are selected using requirements and design specifications and the code of the program under test. This means that the testing team needs access to the internal designs and code for the program.

The chief advantage of white-box testing is that it tests the internal details of the code and tries to check all the paths that a program can execute to determine if a problem occurs. As a result of this white-box test cases can check any additional code that has been implemented but not specified.

[8] COTS stands for **C**ommercial **O**ff **T**he **S**helf

The main disadvantages of white-box testing is that you must wait until after designing and coding the program under test in order to select test cases. In addition, if some functionality of a system is not implemented, using white-box testing may not detect this.

The term "black-box testing" and "white-box testing" are becoming increasingly blurred. For example, many of the white-box testing techniques that have been used on programs, such as control-flow analysis and mutation analysis, are now being applied to the specifications of programs. That is, given a formal specification of a program, white-box testing techniques are being used on that specification to derive test cases for the program under test. Such an approach is clearly black-box, because test cases are selected from the specified behaviour rather than the program, however, the techniques come from the theory of white-box testing.

In these notes, we deliberately blur the distinctions between black-box and white-box testing for this reason.

1.6 Error Guessing

Before we dive into the world of systematic software testing, it is worth mentioning one highly-effective test strategy that is always valuable when combined with any other strategy in these notes. This technique is known as *error guessing*.

Error guessing is an *ad-hoc* approach based on intuition and experience. The idea is to identify test cases that are considered likely to expose errors.

The general technique is to make a list, or better a taxonomy (a hierarchy), of possible errors or error-prone situations and then develop test cases based on the list.

The idea is to document common error-prone or error-causing situations and create a defect history. We use the defect history to derive test cases for new programs or systems. There are a number of possible sources for such a defect history, for example:

- **The Testing History of Previous Programs** — develop a list of faults detected in previous programs together with their frequency;
- **The Results of Code Reviews and Inspections** — inspections are not the same as code reviews because they require much more detailed defect tracking than code reviews; use the data from inspections to create defect lists.

Some examples of common faults include test cases for empty or null strings, array bounds and array arithmetic expressions (such as attempting to divide by zero), and blank values, control characters, or null characters in strings.

Error guessing is not a testing technique that can be assessed for usefulness or effectiveness, because it relies heavily on the person doing the guessing. However, it takes advantage of the fact that programmers and testers both generally have extensive experience in locating and fixing the kinds of faults that are introduced into programs, and can use their knowledge to guess the test inputs that are likely to uncover faults for specific types of data and algorithm.

Error guessing is ad-hoc, and therefore, not *systematic*. The rest of the techniques described in these notes are systematic, and can therefore be used more effectively as quality assurance activities.

1.7 Some Testing Laws

Dijkstra's Law Testing can only be used to show the presence of errors, but never the absence of errors.

Hetzel-Myers Law A combination of different V&V methods out-performs and single method alone.

Weinberg's Law A developer is unsuited to test their own code.

Pareto-Zipf principal Approximately 80% of the errors are found in 20% of the code.

Gutjar's Hypothesis Partition testing, that is, methods that partition the input domain or the program and test according to those partitions, is better than random testing.

Weyuker's Hypothesis The adequacy of a test suite for coverage criterion can only be defined intuitively.

1.8 Bibliography

[1] G.J. Myers, *The Art of Software Testing*, John Wiley & Sons, 1979.

[2] D.E. Knuth, *The Art of Computer Programming, vol. 2: Semi-numerical Algorithms*, 2nd Ed., Addison Wesley, 1981.

[3] B. Beizer, *Software Testing Techniques*, 2nd ed., van Nostrand Reinhold, 1990.

[4] E. Kit, *Software Testing in the Real World*, Addison-Wesley, 1995.

[5] R. Hamlet, Random Testing, In *Encyclopedia of Software Engineering*, J. Marciniak ed., pp. 970–978, Wiley, 1994.

[6] A. Endres and D. Rombach, *A Handbook of Software and Systems Engineering*, Addison-Wesley, 2003.

[7] J. A. Whittaker, *How to Break Software: A Practical Guide to Testing*, Addison-Wesley, 2002.

Chapter 2

Input Partitioning

Recall from Section 1.4, that each program has an *input domain* — the set of values that can be used as input to the program. Test input selection is all about selecting the values in this domain that have the highest likelihood of producing failures.

Using input partitioning, programs are considered as the composition of an *input classifier*, that classifies the input domain into one of a number of different classes where each input class computes one function of the overall program.

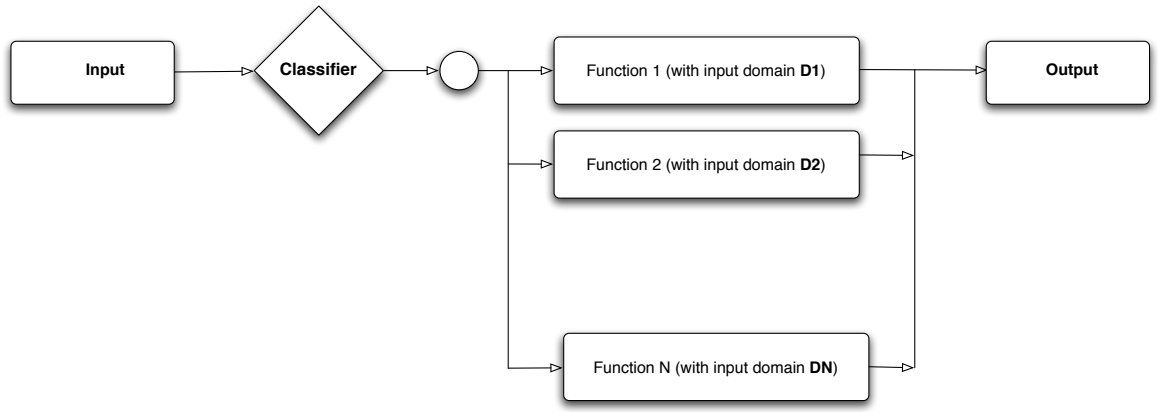


Figure 2.1: The Domain Model: The program acts as a classifier (or filter) that takes each input and determines which of the many program functions to apply to the input.

The canonical representation of a program is the set $\{(D_1; F_1), (D_2; F_2), \ldots\}$, where D_i is the i^{th} domain and F_i is the corresponding function computed by the program. There is a one-to-one correspondence between the input domains and the functions computed by a program. Each of the functions computed by a program occurs along a *program path*, that is, a path in the program that executes a sequence of statements for computing the function. In domain testing, an input domain is the subset of all inputs that will trigger a specific program function to be computed along a specific *program path*.

As an example, consider the program in Figure 2.2, which calculates the minimum of two integers, `x` and `y`. There are two paths in this program. The first executes the statements at lines 3, 4, 5, and 7, while the second executes at lines 3, 4, and 7. Each of these can be conceived to be its own function. The first one returns the value of `y`, while the second returns the value of `x`. The input classifier executes the first function if `x` is greater than `y`, and executes the second otherwise.

Based on this view of a program, there are two possible types of faults:

```
1.    int min(int x, int y)
2.    {
3.        int minimum = x;
4.        if (x > y) {
5.            minimum = y;
6.        }
7.        return minimum;
8.    }
```

Figure 2.2: An implementation of the Min function

- *Computation faults*: where the correct path is chosen but an incorrect computation occurs along that path; and
- *Domain faults*: where the computation is correct for each path but an incorrect path is chosen.

  The possible causes for an incorrect path are: (1) the incorrect path is executed for the input domain; (2) the decisions that make up the path selection may contain a fault; and (3) the correct path (or a fragment of a path necessary for the computation) may simply be missing.

The aim of input partitioning is to derive test inputs that exercise each function of the program at least once. In this chapter, we present two methods for partitioning the input into *equivalence classes*.

2.1 Equivalence Classes

In functional testing, we can systematically derive equivalence classes and test inputs that are more likely to find faults than by using random testing. Using these techniques, we have an better idea of what constitutes an equivalence class.

> **Definition 4** An *equivalence class* is a set of values from the input domain that are considered to be as likely as each other to produce failures for a program. Relating this to Figure 2.1, there are N equivalence classes: one for each function. Intuitively, this says that, for an equivalence class, each element in that equivalence class should execute the same statements in a program in the same order, but just with different values.

In other words, a single test input taken from an equivalence class is representative of *all* of the inputs in that class.

Recall the Min function from Figure 2.2. In that program, all inputs that satisfy the relation $\mathsf{x} > \mathsf{y}$ will execute one path (inside the if statement), while the rest, which satisfy the relation $\mathsf{x} \leq \mathsf{y}$, will execute another (not executing inside the if statement). Therefore, all values that satisfy $\mathsf{x} > \mathsf{y}$ form one equivalence class, while the rest form another.

Input conditions, valid inputs, and invalid inputs

Each equivalence class is used to represent certain *input conditions* on the input domain. Equivalence partitioning tries to break up input domains into sets of *valid* and *invalid* inputs based on these input conditions. Unfortunately, the terms "input condition", "valid" and "invalid" are not always used consistently but the following definition will be used in these notes to give a consistent meaning of the terms for this subject.

> **Definition 5**
>
> (i) An *input condition* on the input domain is a predicate on the values of a single variable in the input domain. When we need to consider the values of more than one variable in the input domain, we refer to this as the *combination of input conditions*.

(ii) A *valid* input to a program is an element of the input domain that is the value expected from the program specification; that is, a non-error value according to the program specification.

(iii) An *invalid* input is an input to a program is an element of the input domain that is not expected or an error value that as given by the program specification. In other words, an invalid input is a value in the input domain that is not a valid input.

The class of invalid inputs can be further broken down into two sub-classes: those that are *testable*; and those that are *not* testable.

A non-testable input is one that violates the *precondition* of a program. A precondition is a condition on the input variables of a program that is *assumed to hold* when that program is executed. This means that the programmer has made an explicit assumption that the precondition holds at the start of the program, and as such, the behaviour of the program for any input that violates the precondition is *undefined*. If the behaviour is undefined, then *we do not test for it*.

A testable invalid input is an invalid input that does not violate the precondition. Typically, these refer to inputs that return error codes or throw exceptions.

The space of inputs can be modelled as in Figure 2.3.

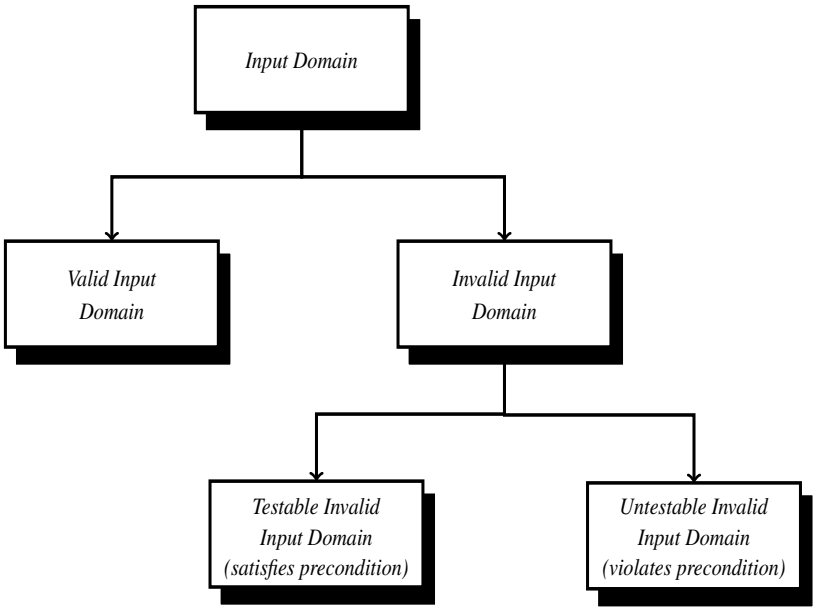


Figure 2.3: Hierarchy of input domains

Example 6 *As an example of the above, consider a* binary search *function, which, given a* sorted *list of numbers, determines whether or not a specific number is in that list. The efficiency of a binary search relies on the fact that the input list is sorted. Thus, the precondition of the function is that the list is sorted. There is little point writing tests with unsorted lists, because the behaviour of a binary search is undefined for an unsorted list.*

The input condition on the list is a predicate specifying that the list is sorted. Any sorted list is a valid input, while any unsorted list is an invalid input.

Partioning and equivalence classes

Input partitioning is then a systematic method for identifying interesting input conditions to be tested. To make our use of the term input condition consistent with other literature we will assume that an input condition can be applied to set of values of a specific input variable, or a set of input variables as well.

The assumption behind equivalence partitioning is that all members of the class behave in the same way with respect to failures, so choosing one member of an equivalence class has the same likelihood of detecting a failure as any other member of the equivalence class. This assumption is not correct, but nonetheless, partitioning the input domain into equivalence classes is a valuable technique for testing when combined with other techniques.

There are two key properties equivalence classes for software testing. If we have n equivalence classes $\mathsf{EC_1}, \ldots, \mathsf{EC_n}$ for an input domain, ID, the following two properties must hold:

- for any two equivalence classes, $\mathsf{EC_i}$ and $\mathsf{EC_j}$, such that $\mathsf{i} \neq \mathsf{j}$, $\mathsf{EC_i} \cap \mathsf{EC_j} = \emptyset$; that is, $\mathsf{EC_i}$ and $\mathsf{EC_j}$ are *disjoint*; and
- $\bigcup\{\mathsf{EC_1}, \ldots, \mathsf{EC_n}\} = \mathsf{ID}$; that is, the testable input domain is covered.

Together, these two properties say that the equivalence classes are disjoint and they cover the input domain. In other words, for any value in the input domain, that input belongs to *exactly one* equivalence class.

The key question to answer now is: *what constitutes a good equivalence class*? This chapter presents two systematic techniques for partitioning the input domain of a program into equivalence classes. The first one relies on the underlying structure of the program to determine which inputs execute the same program statements — therefore, it is a *white-box* testing technique. The second technique relies on the specified functionality of the program — therefore, it is a *black-box* testing technique. To avoid confusion between the two, we will refer to the former as *domain testing*, and the latter as *equivalence partitioning*.

These two techniques are related, but the underlying concepts are different. Domain testing determines the *actual* equivalence classes of a program, while equivalence partitioning, on the other hand, determines the *expected* equivalence classes of a program from its functional requirements.

2.2 Domain Testing

In this section, we present *domain testing*. Domain testing partitions the input of a program using the program structure itself. The test inputs are executed on the program, and the tester evaluates whether the output of the program is in the expected output domain.

To select test cases according to the domain testing strategy, the input space is first partitioned into a set of mutually exclusive equivalence classes. Each equivalence class corresponds to a program path. The idea is to select test cases based on the domain boundaries and to explore domain boundaries in order to work out whether or sub-domains are correct or not. More precisely, test cases are selected from: (1) the domain boundaries; and (2) points close to the domain boundaries.

We will need some standard terminology for domain testing.

- **Relational Expressions**: are of the form $A\ rel\ B$, where rel denotes a relational operator, that is, one of $>$, $<$, $\geq$, $\leq$, $=$ and $\neq$.
- **Predicates**: are points in a program where the control flow can diverge.
- **Simple Predicates**: are single relational expressions.
- **Compound Predicates**: are predicates composed of several simple predicates combined by Boolean operators such as AND ($\wedge$), OR ($\vee$) and NOT ($\neg$).
- **Linear Simple Predicates**: are simple predicates of the form $(a_1x_1 + \ldots + a_nx_n)\ \ rel\ \ k$, where rel is a relational operator, $x_1, \ldots, x_n$, and the a_is and k are constants.
- **Path Condition**: is the condition that must be satisfied by the input data for that path to be executed.

Example 7

```
int f(float x, y)
{
    float w, z;

    w = x + y;

    if (x + y >= 2) {
        z = x - (3 * y);
    }
    else {
        z = 2x - 5y;
    }

    if (z + w > 3) {
        z = f(x, y, z);
    }
    else {
        z = g(x, y, z);
    }

    return z;
}
```

Figure 2.4: A small (non-sensical) program to illustrate the idea of domains.

As an example consider the program in Figure 2.4. The domains of this program are calculated by looking at the different paths in the program. In this example, there are two if/then/else statement, which gives us four possible paths. Table 2.1 outlines the four path conditions for each path, which correspond to the predicate in the branches.

Considering that a domain is the set of inputs that satisfy a path condition, then four path conditions gives us four domains. Each domain is defined by a *linear border*. These are shown Figure 2.5. In this figure, TT refers to the domain in which both branches in the program evaluate to true. TF, TF, and FF are defined similarly to other possible evaluations.

C1a		x + y $\geq$ 2
C1b		x + y $<$ 2
C2a	(C1 T)	2x - 2y $>$ 3
C2b	(C1 F)	3x - 4y $>$ 3

Table 2.1: The four domains from the example program.

If the branches of a program are such that certain paths are *infeasible*, then the domain for that path is empty.

Remark 8 We note that domain testing makes an assumption that programs are *linearly domained*. Linear equalities are expressions that can be put into the form

$$a_1x_1 + \ldots + a_nx_n = k \text{ or } a_1x_1 + \ldots + a_nx_n \neq k$$

and linear inequalities are expressions that can be put into the form

$$a_1x_1 + \ldots + a_nx_n \leq k \text{ or } a_1x_1 + \ldots + a_nx_n \geq k$$

or

$$a_1x_1 + \ldots + a_nx_n < k \text{ or } a_1x_1 + \ldots + a_nx_n > k.$$

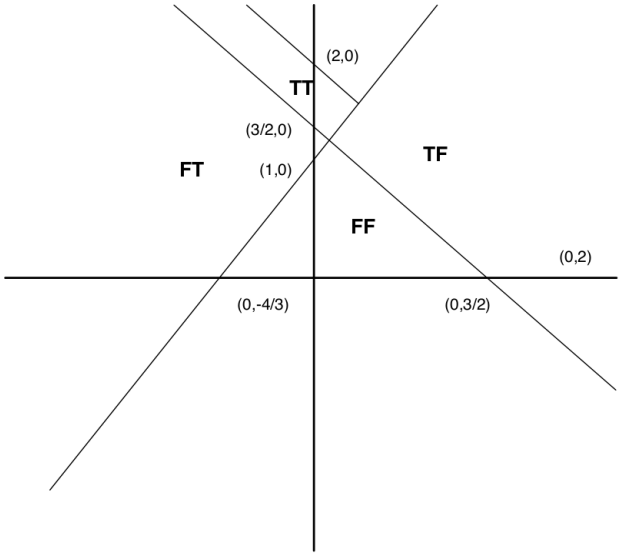


Figure 2.5: The four domains from the program.

Linear borders require fewer test points to check and can be checked with more certainty.

If a program violates this assumption then testing using this strategy may not be as productive at finding faults as would have been the case if the program did satisfy the assumptions, because solving non-linear equalities and inequalities is considerably more difficult that solving linear equalities and inequalities.

In the non-linear case a point may lie in a domain but be computationally too expensive to check accurately; for example, a polynomial:

$$a_1 {x_1}^y + \ldots + a_n x_n \leq k$$

Compound Predicates

Simple predicates contain just a set of variables and a relational operator such as $\leq$, $\geq$, $>$, $<$, $=$ or $\neq$. Simple predicates create *simple boundaries*. When we use *compound predicates* to define boundaries then the situation becomes trickier to test. A compound predicate with a logical conjunction (**AND** operator) is straight-forward because it often defines a convex domain where all of the points lie within a boundary. A compound predicate with a logical disjunction (**OR** operator) can create disjointed boundaries, for example, $x \leq 0 \vee x \geq 2$ creates the boundaries as in Figure 2.6.

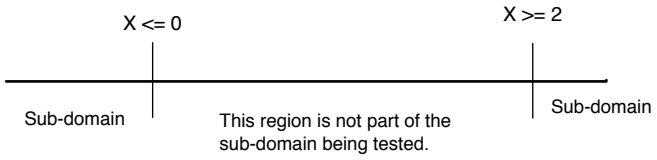


Figure 2.6: A boundary defined by a compound disjunction.

To overcome this problem, domain testing treats compounds predicates such as logical disjunction as different paths. That is, in the example in Figure 2.6, the cases of $x \leq 0 \vee x \geq 2$ are treated as two separate functions. Each compound predicate is broken into *disjunctive normal form*; that is, a disjunction of one or more simple predicates.

To do this, each compound predicate is reduced to a predicate using only conjunction and negation, using *DeMorgan's Laws*. For example, using the law

$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$

we can reduce the expression $\neg(\mathrm{x} \leq 0 \wedge \mathrm{x} \geq 2)$ to $x > 0 \vee x < 2$.

Loops in Programs

If each program path corresponds to a function, then for some programs containing loops, we have an infinite number of paths, and therefore, an infinite number of equivalence classes. Selecting test inputs for each of these classes is clearly impossible, so we need a technique for making this finite.

A typical work around is to apply the *zero-to-many* rule. The zero-to-many rule states that the minimum number of times a loop can execute is zero, so this is one path. Similarly, some loops have an upper bound, so treat this as a path. A general guideline for loops is to select test inputs that execute the loop:

- zero times – so that we can test paths that do not execute the loop;
- once – to test that the loop can be entered and that the results for a single iteration are correct;
- twice – to test that the results remain correct between different iterations of the loop;
- N times (greater than 2) – to test that an arbitrary number of iterations returns the correct results; and
- N+1 times to test that after an arbitrary number of iterations the results remain correct between iterations.

In the case that we know the upper bound of the loop, then set N+1 to be this upper bound. Therefore, we are partitioning the input domain based on the number of times that a loop executes. We need only to instantiate the number, N, when selecting the test inputs that satisfy the equivalence classes.

Example 9 *Selecting Test Cases for the Triangle Program*

Consider a program, which takes as input, three integers, `x`, `y`, and `z`, each representing the length of one side of a triangle, and classifies the triangle as one of the following:

(i) *invalid*: at least one side does not have a positive length, or one side is greater then the sum of the other two sides;

(ii) *equilateral*: all sides are the same length;

(iii) *isosceles*: two sides are the same length; or

(iv) *scalene*: are sides are of different length.

An implementation of this program is shown below.

```
type enum Triangle = {equilateral, isosceles, scalene, invalid}

Triangle categorise(int x, int y, int x)
{
   if (x > 0 && y > 0 && z > 0 &&
       x + y >= z && x + z >= y && y + z >= x) {
       if (x = y)
           if (y = z) return equilateral;
           else return isosceles;
       else if (y = z) return isosceles;
       else if (x = z) return isosceles;
       else return scalene;
   }
   else return invalid;
}
```

To derive the paths for this program, we look at the branches in the program source. For simplicity, we abbreviate the predicate in the first branch to valid(x, y, z).

The valid paths in the program as straightforward to identify. The equivalence class that corresponds to an equilateral triangle is characterised by the predicate $\mathsf{valid}(\mathsf{x},\mathsf{y},\mathsf{z}) \wedge \mathsf{x} = \mathsf{y} \wedge \mathsf{y} = \mathsf{z}$. For the isosceles case, there are three equivalence classes:

(i) $\mathsf{valid}(\mathsf{x},\mathsf{y},\mathsf{z}) \wedge \mathsf{x} = \mathsf{y} \wedge \mathsf{y} \neq \mathsf{z}$

(ii) $\mathsf{valid}(\mathsf{x},\mathsf{y},\mathsf{z}) \wedge \mathsf{x} \neq \mathsf{y} \wedge \mathsf{y} = \mathsf{z}$

(iii) $\mathsf{valid}(\mathsf{x},\mathsf{y},\mathsf{z}) \wedge \mathsf{x} \neq \mathsf{y} \wedge \mathsf{y} \neq \mathsf{z} \wedge \mathsf{x} = \mathsf{z}$

Finally, the equivalence class for the scalene triangle is $\mathsf{valid}(\mathsf{x},\mathsf{y},\mathsf{z}) \wedge \mathsf{x} \neq \mathsf{y} \wedge \mathsf{y} \neq \mathsf{z} \wedge \mathsf{x} \neq \mathsf{z}$

The invalid case is more difficult to partition. The false condition of the first branch is taken if $\neg\mathsf{valid}(\mathsf{x},\mathsf{y},\mathsf{z})$, which is equivalent to

$$\neg(x > 0 \wedge y > 0 \wedge z > 0 \wedge x + y \geq z \wedge x + z \geq y \wedge y + z \geq x)$$

Using DeMorgan's laws to reduce this into disjunctive normal form, we are left with six cases: the negation of each of the simple predicates above.

Combining the invalid with the valid cases gives us 11 abstract test cases, which must be then instantiated with actual values. For example, choose the test inputs $\mathsf{x} = 3 \wedge \mathsf{y} = 3 \wedge \mathsf{z} = 3$ for the equilateral case.

2.3 Equivalence Partitioning

In this section, we present *equivalence partitioning*. Equivalence partitioning partitions the input of a program using the functional requirements of the program. The test inputs are executed on the program, and the tester evaluates whether the output of the program is in the expected output domain.

Equivalence partitioning is one of the oldest and still most widely used method for selecting test cases based on a partitioning of the input domain.

A Method for Choosing Equivalence Classes

The aim is to minimise the number of test cases required to cover all of the equivalence classes that you have identified. The general method only needs three steps:

A. Identify the initial equivalence classes (ECs);

B. Identify overlapping equivalence classes, and eliminate them by making the overlapping part a new equivalence classes; and

C. Select one element from each equivalence class as the test input and work out the expected result for that test input.

Step 1 - Identify the initial equivalence classes

Unfortunately, there is no clear cut algorithm or method for choosing test inputs according to equivalence partitioning. You will still need to build up some judgement and intuition. There are however, a good set of guidelines to get you going. Again, the guidelines are just suggestions for picking good equivalence classes – the guidelines work in many cases but there are always exceptions where you will need to exercise your own creativity.

The following set of guidelines will help you to identify *potential* equivalence classes. As you test a program you may need to choose extra test cases in order to explore different subsets of the input domain more thoroughly. The more you apply the guidelines and see the effect, the easier it will be for you to start choosing good classes for testing - practice will give you insight and experience.

Guidelines

1. If an input condition specifies a range of values, identify one valid equivalence class for the set of values in the range, and two invalid equivalence classes; one for the set of values below the range and one for the set of values above the range.

   Example 10 *If we are given the range of values* $1 \ldots 99$*, then we require three equivalence classes:*

   - *The valid equivalence class* $1 \ldots 99$*; and*
   - *The two invalid equivalence classes* $\{x|x < 1\}$ *and* $\{x|x > 99\}$.

2. If an input condition specifies a set of possible input values and each is handled differently, identify a valid equivalence class for each element of the set and one invalid equivalence class for the elements that are not in the set.

   Example 11 *If the input is selected from a set of vehicle types, such as* $\{BUS, TRUCK, TAXI, MOTORCYCLE\}$*, then we require:*

   - *Four valid equivalence classes; one for each element of the set* $\{BUS, TRUCK, TAXI, MOTORCYCLE\}$*; and*
   - *One invalid equivalence class for an element "outside" of the set; such as* $TRAILER$ *or* $NON\_VEHICLE$.

3. If the input condition specifies the number (say N) of valid inputs, define one valid equivalence class for the correct number of inputs and two invalid equivalence classes – one for values $< N$ and one for values $> N$.

   Example 12 *If the input condition specifies that the user should input exactly three preferences, then define one valid equivalence class for three preferences, and two invalid equivalence classes: one for* < 3 *and one for* > 3.

4. If the input condition specifies that an input is a collection of items, and the collection can be of varying size; for example, a list or set of elements; define one valid equivalence class for a collection of size 0, one valid equivalence class for a collection of size 1, and one valid equivalence class for a collection of size > 1.

   This is sometimes called the *zero-one-many* rule.

   Additionally, if the collection has *bounds*, such as a maximum number N, define one valid equivalence class for a collection of size N, and one invalid equivalence class for a collection of size $> N$. Similarly for lower bounds, if they exist.

   Example 13 *If the input condition specifies that the user should input a list of their preferences, with no bound on the list, then define valid equivalence classes for lists of size 0, 1, and 5.*

   If the input condition is further constrained by a maximum size of 10, then define valid equivalence classes for lists of size 0, 1, 5, and 10, as well as an invalid equivalence class for a list of size 11.

5. If an input condition specifies a "must be" situation, identify one valid equivalence class and one invalid equivalence class.

   Example 14 *If the first character of an input* must be *a numeric character, then we require two equivalence classes – a valid class*

   $$\{s|\text{the first character of } s \text{ is a numeric}\}$$

   and one invalid class

   $$\{s|\text{the first character of } s \text{ is not a numeric}\}$$

6. If there is any reason to believe that elements in an equivalence class are handled in a different manner than each other by the program, then split the equivalence class into smaller equivalence classes.

   This is a sort of "default" catch statement, which really says that you can derive tests based on intuition or your understanding of the program and domain, where none of the other guidelines fit.

Notice that this particular set of guidelines has some very general rules such as the last one. In short, you will need to build up some judgement and experience in order to become good at selecting test cases.

Step 2 - Eliminating overlapping equivalence classes

If we apply the guidelines in Step 1, it is likely that we will end up with overlapping equivalence class. Figure 2.7 shows an input domain that has two equivalence classes that overlap.

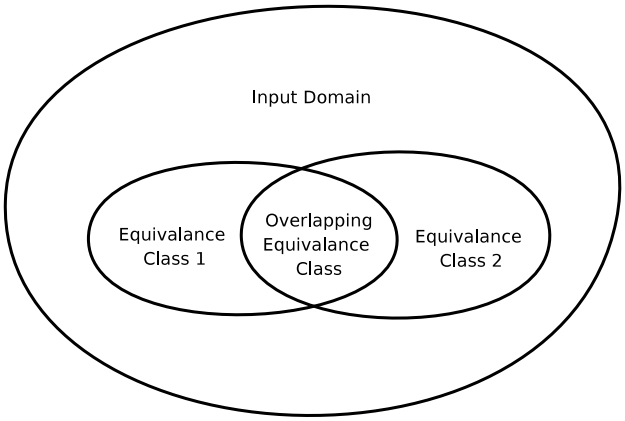


Figure 2.7: Overlapping equivalence classes on an input domain.

Returning to our example of a binary search function, we may identify the following equivalence classes (in fact, we should identify even more):

A. an empty list;
B. a list with exactly one element;
C. a list with at least two elements;
D. a sorted list; and
E. a unsorted list (an invalid input).

Equivalence classes A, B, and C are disjoint, as are D and E. However, a list with at least two elements can both sorted or unsorted, so there is overlap between classes C and D, as well as C and E.

Once these overlapping classes has been identified, they are eliminated by treating them as equivalence classes themselves. So, for the equivalence classes in Figure 2.7, the overlapping class becomes an equivalence class of its own, and the other two equivalence classes are re-calculated by removing the values in the overlapping class.

In the binary search example, we identify the overlap, and re-calculate the new classes. As it turns out, a list with two or more elements must be either sorted or unsorted, so the set of values in class C is equal to the set of values in the union of D and E. As a result, we end up with the following equivalence classes:

1. an empty list;
2. a list with exactly one element;
3. a unsorted list with at least 2 elements;
4. a sorted list with at least 2 elements;

In this example, we end up with *less* equivalence classes, however, in other cases we can end up with more.

Step 3 - Selecting test cases

Once the equivalence classes have been identified, selecting the test cases is straightforward. Any value from an equivalence class is identified to be as likely to produce a failure as any other value in that class, therefore *any element of the class serves as a test input*, and selecting any arbitrary element from the class is adequate.

Example 15 *Selecting Test Cases for the Triangle Program*

Recall the triangle program from Example 9. Consider the following *informal specification* for that program:

> The program reads three integer values from the standard input. The three values are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle is
>
> (i) *invalid*: at least one side does not have a positive length, or one side is greater then the sum of the other two sides;
>
> (ii) *equilateral*: all sides are the same length;
>
> (iii) *isosceles*: two sides are the same length; or
>
> (iv) *scalene*: are sides are of different length.

The triangle classification program requires three inputs to the program as well as the form of the output. Further we require that each of the inputs "must be" a positive integer. Table 2.2 summarises the input conditions.

Requirement	**Input Condition**
Input three integers	Let the three integers be x, y and z; then $\mathsf{x} \in \mathsf{int}$, $\mathsf{y} \in \mathsf{int}$ and $\mathsf{z} \in \mathsf{int}$ are the input conditions.
Integers must be positive	$\mathsf{x} > 0$, $\mathsf{y} > 0$ and $z > 0$

Table 2.2: The input conditions for the triangle classification program.

Now we can use the guidelines and the specification to determine valid and invalid equivalence classes for the input conditions. Guideline 1 is appropriate as we have a range of values. If the three integers we have called x, y and z are all greater than zero then there are three valid equivalence classes:

$$\begin{aligned} \mathsf{EC}_{\mathsf{valid}_1} &= \{\mathsf{x} | \mathsf{x} > 0\} \\ \mathsf{EC}_{\mathsf{valid}_2} &= \{\mathsf{y} | \mathsf{y} > 0\} \\ \mathsf{EC}_{\mathsf{valid}_3} &= \{\mathsf{z} | \mathsf{z} > 0\} \end{aligned}$$

for the input conditions $\mathrm{x} > 0$, $\mathrm{y} > 0$ and $\mathrm{z} > 0$. However, these are not disjoint. After eliminating the overlapping cases, we are left with only one equivalence class:

$$\mathsf{EC}_{\mathsf{valid}} = \{(\mathsf{x}, \mathsf{y}, \mathsf{z}) | \mathsf{x} > 0 \wedge \mathsf{y} > 0 \wedge \mathsf{z} > 0\}$$

Adding the constraint that each side cannot be greater than the sum of the other two sides, and eliminating the equivalence classes leaves us with only one equivalence class again:

$$\mathsf{EC}_{\mathsf{valid}} = \{(\mathsf{x}, \mathsf{y}, \mathsf{z}) | \mathsf{x} > 0 \wedge \mathsf{y} > 0 \wedge \mathsf{z} > 0 \wedge \mathsf{x} + \mathsf{y} \geq \mathsf{z} \wedge \mathsf{x} + \mathsf{z} \geq \mathsf{y} \wedge \mathsf{y} + \mathsf{z} \geq \mathsf{x}\}$$

The output domain consists of $\mathsf{isosceles}$, $\mathsf{scalene}$, $\mathsf{equilateral}$ and $\mathsf{invalid}$. Now we will need to apply guideline 3 because different values in the input domain map to different elements of the output domain. For example, the equivalence class for equilateral triangles would be

$$\mathsf{EC}_{\mathsf{equilateral}} = \{(\mathsf{x}, \mathsf{y}, \mathsf{z}) | \mathsf{x} = \mathsf{y} = \mathsf{z}\}.$$

If we create one equivalence class for each of the different values in the output domain, we split the initial equivalence classes into three disjoint classes.

For the invalid classes we need to consider the case where each of the three variables in turn can be negative, or

longer than the other two sides:

$$
\begin{aligned}
\mathsf{EC}_{\mathsf{Invalid}_1} &= \{\mathsf{x} | \mathsf{x} \leq 0\} \\
\mathsf{EC}_{\mathsf{Invalid}_2} &= \{\mathsf{y} | \mathsf{y} \leq 0\} \\
\mathsf{EC}_{\mathsf{Invalid}_3} &= \{\mathsf{z} | \mathsf{z} \leq 0\} \\
\mathsf{EC}_{\mathsf{Invalid}_4} &= \{(\mathsf{x}, \mathsf{y}, \mathsf{z}) | \mathsf{x} + \mathsf{y} \leq \mathsf{z}\} \\
\mathsf{EC}_{\mathsf{Invalid}_5} &= \{(\mathsf{x}, \mathsf{y}, \mathsf{z}) | \mathsf{x} + \mathsf{z} \leq \mathsf{y}\} \\
\mathsf{EC}_{\mathsf{Invalid}_6} &= \{(\mathsf{x}, \mathsf{y}, \mathsf{z}) | \mathsf{y} + \mathsf{z} \leq \mathsf{x}\}
\end{aligned}
$$

Again, there is overlap here. It is possible that both x and y can be less than 0. So, the overlapping cases need to be eliminated. This is problematic, because there are a number of combinations here. This is discussed further in Section 2.5. For now, we will add just one test case from each of the (overlapping) classes above.

According to the equivalence partitioning method we only need to choose one element from each of the classes above in order to test the triangle program. Table 2.3 gives the test cases that are obtained after **Step 3** has been completed.

Equivalence class	**Test Input**	**Expected Outputs**
$\mathsf{EC}_{\mathsf{equilateral}}$	(7, 7, 7)	equilateral
$\mathsf{EC}_{\mathsf{isosceles}}$	(2, 3, 3)	isosceles
$\mathsf{EC}_{\mathsf{scalene}}$	(3, 5, 7)	scalene
$\mathsf{EC}_{\mathsf{invalid}_1}$	(-1, 2, 3)	invalid
$\mathsf{EC}_{\mathsf{invalid}_2}$	(1, -2, 3)	invalid
$\mathsf{EC}_{\mathsf{invalid}_3}$	(1, 2, -3)	invalid
$\mathsf{EC}_{\mathsf{invalid}_4}$	(1, 2, 5)	invalid
$\mathsf{EC}_{\mathsf{invalid}_5}$	(1, 5, 2)	invalid
$\mathsf{EC}_{\mathsf{invalid}_6}$	(5, 2, 1)	invalid

Table 2.3: The test cases for the triangle program.

Selecting just *any* value in an equivalence class may not be *optimal* for finding faults. For example, consider the faulty implementation of the square function shown in Figure 1.4, Section 1.3.2. Recall from the discussion in Section 1.3.2 that the test input 2 for this function would not reveal a failure due to coincidental correctness. In the absence of any other information about the square function such as the maximum integer input or the largest possible result, the input domain and output domains for square are simply int and the valid equivalence class is some bounded subset of int. Selecting a test input of 2 does not reveal a fault in the function, but the tester cannot know this.

Is there a better way to select test inputs from equivalence classes to give a greater likelihood of finding faults?

Clearly, we can select *more than one input* from an equivalence class, which reduces the chances of coincidental correctness, but increases the size of our test suite. Hierons [2] discusses how, in some cases, coincidental correctness in *boundary-value analysis* (see Chapter 3) can be avoiding using information in the specification. However, his ideas do not work for the square function.

Example 16 *Selecting Test Cases for the GetWord Function*

Consider the following specification of a function, GetWord, that returns the next word in a string:

(1) The GetWord function must accept a string and return two items: (1) the first word in the string; and (2) the string with the first word removed.

(2) Words are considered to be a string of characters not containing a blank character.

(3) The blank characters between words can be spaces, tabs, or newline characters.

(4) Strings can be a maximum of 200 characters in length.

An algorithm for GetWord is given in Figure 2.8.

There are at least three steps in implementing a specification:

Step 1 choose a design that will implement all of the requirements in the specification;

Step 2 choose algorithms and data structures that implement the functionality in the design;

Step 3 write the actual program, using some programming, that implements the algorithm.

In step 2 and step 3 there are choices and sometimes the programming language constructs that are chosen to implement an algorithm can introduce subtle behaviours that the specification writer did not anticipate.

The algorithm in Figure 2.8 makes some *design choices* about how to achieve the requirements. Requirement (1) requires a function that takes strings and returns a pair of strings: one for the word, and one for the rest of the line.

The algorithm implements this requirement by a function that takes a string as an input argument, returns a string (the first word), and the input string (the parameter line line) to return the input with first word removed. Doing this is a distinct design choice.

```
1. string GetWord(string line)
2.         while ( line[next] is a blank character) do
3.                 next = next + 1;
4.         if ( line[next] is not the end of the line ) then
5.                 while ( line[next]) is not a blank character ) do
6.                         word[i] = line[next];
7.                         next = next + 1;
8.                         i = i + 1;
9.                 return word;
10.                line = line[next...length(line)];
11.        else
12.                return "";
```

Figure 2.8: The algorithm for the GetWord function.

Equivalence partitioning requires only the specification of the program to derive test cases. This makes it ideal for system level testing and integration level testing. However, there are times when we have more information available to us than just the specification. Sometimes we have an algorithm that can be used to define "better" input conditions than those given by the specification alone, and to choose better equivalence classes.

Let us assume that the input domain for the GetWord function is the set of strings of length 200 characters or less. The output domain is the set of pairs strings of 200 characters or less. Note that, as the input string is a maximum of 200 characters, any of its substrings will be a maximum of 200 characters in length.

Using the guidelines from Section 2.3 in combination with the specification for GetWord, we now have an input condition that the length of the input string must be in the range $0 \ldots 200$ characters. Using guideline 1 we have one valid equivalence class and two invalid classes; the valid class consists of strings of length between $0 \ldots 200$ and an invalid class of length > 200. The other invalid class consists of strings of length < 0 which is infeasible because we cannot supply a string with a negative length. So we have:

$$\begin{aligned} \mathsf{EC}_{\mathsf{valid}} &= \{\, \mathsf{s} \mid \mathsf{s} \text{ is a string and } \mathsf{length}(\mathsf{s}) \leq 200 \,\} \\ \mathsf{EC}_{\mathsf{invalid}} &= \{\, \mathsf{s} \mid \mathsf{s} \text{ is a string and } \mathsf{length}(\mathsf{s}) > 200 \,\} \end{aligned}$$

But we are not yet done. The class $\mathsf{EC}_{\mathsf{valid}}$ can be partitioned into smaller classes — the algorithm in Figure 2.8 specifies a number of possibilities. This is the creative part of testing — thinking about what can go wrong and finding test cases that show that a program *does* go wrong.

Strings can be empty, they consist of one word or multiple words, there can be a single blank between words or a number of blanks between words, and words can contain legal characters or illegal characters. Using this information we can decompose $\mathrm{EC}_{\mathrm{valid}}$ into smaller ECs as follows.

$$
\begin{aligned}
\mathsf{EC}_{\mathsf{valid}_1} &= \{\mathsf{s} \mid \mathsf{s} \text{ is empty}\} \\
\mathsf{EC}_{\mathsf{valid}_2} &= \{\mathsf{s} \mid \mathsf{s} \text{ has a single word}\} \\
\mathsf{EC}_{\mathsf{valid}_3} &= \{\mathsf{s} \mid \mathsf{s} \text{ has more than one word}\} \\
\mathsf{EC}_{\mathsf{valid}_4} &= \{\mathsf{s} \mid \text{words are separated by single blank}\} \\
\mathsf{EC}_{\mathsf{valid}_5} &= \{\mathsf{s} \mid \text{words are separated by 2 or more blanks}\}
\end{aligned}
$$

Notice that the union of all of the smaller ECs covers our original valid EC, that is, $\bigcup_{\mathrm{i}} \mathrm{EC}_{\mathrm{valid}_{\mathrm{i}}} = \mathrm{EC}_{\mathrm{valid}}$.

As with the triangle function in Example 15, there is overlap in the equivalences classes. In the GetWord case, strings with blanks must also have more than one word, so $\mathsf{EC}_{\mathsf{valid}_3}$ is made redundant by $\mathsf{EC}_{\mathsf{valid}_4}$ and $\mathsf{EC}_{\mathsf{valid}_5}$.

The test cases that we obtain from these equivalence classes are given in Table 2.4. The invalid class consists of strings of more than 200 words in length.

Equivalence Class	**Input**	**Expected Output** (word, line)
$\mathrm{EC}_{\mathrm{valid}_1}$	“”	(“”, “”)
$\mathrm{EC}_{\mathrm{valid}_2}$	“word”	(“word”, “”)
$\mathrm{EC}_{\mathrm{valid}_4}$	“ two words”	(“two”, “ words”)
$\mathrm{EC}_{\mathrm{valid}_5}$	“ two words”	(“two”,“ words”)
$\mathrm{EC}_{\mathrm{invalid}}$	“ Story time . . . ” (200+ chars)	Error value

Table 2.4: The test cases obtained from the equivalence classes.

2.4 Test Template Trees

One complication of using equivalence partitioning is detecting and eliminating overlapping equivalence classes. However, taking a more structured approach to this using *test template trees*, we can avoid this problem by not introducing overlaps in the first place.

A test template tree is an overview of an equivalence partitioning, but importantly, it is *hierarchical*. When applied correctly, it provides a graphical overview of equivalence classes and their *justification*, as well as avoiding overlap.

Process

The process for deriving a test template tree for a program is similar to doing equivalence partitioning.

Start We start with the testable input domain, and aim to break this into smaller equivalence classes that make good tests.

Repeat At each step, we choose an equivalence partitioning guideline to apply to one or more *leaf nodes* in test template tree, breaking the leaf nodes into multiple new leaf nodes, each once more specific than its parent node, which is no longer a leaf node.

When we partition each leaf node, we ensure that the partitioning is: (a) disjoint — the partitions do not overlap; and (b) they partitions cover their parent — that is, if we combine the new partitions, that combination will be

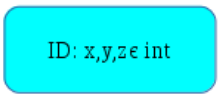
equivalent to their parent. These two properties are the same as listed in Section 2.1 for equivalence classes. By avoiding creating overlap and by covering each parent, we ensure (inductively) that the resulting equivalence classes at the end of this entire process also do not overlap and their cover the root node (the input domain).

End The process ends when either: (a) there are no more sensible partitionings to apply; or (b) our tree grows too large and we can no longer manage the complexity (although there are ways to mitigate this as we will discuss soon).

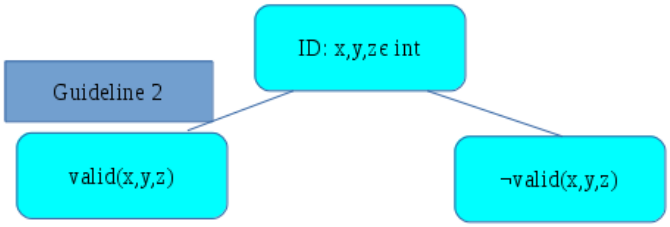
Result To construct the equivalence classes, we simply take each leaf node and derive the equivalence from that lead node to the root of the tree.

The resulting equivalence classes do not overlap and they cover the entire input domain. This is because, at each partition, we ensure this property held locally. This local property means that non-overlap and coverage also hold globally, which can be demonstrated inductively.

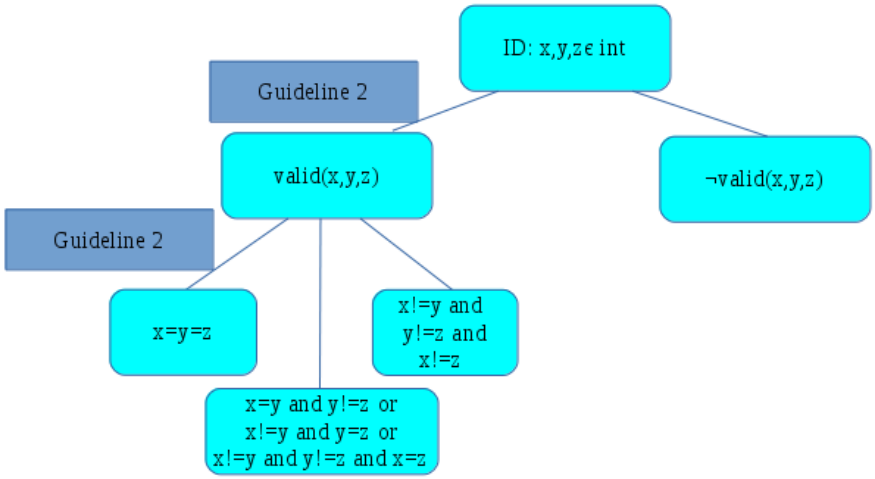
Example 17 *As an example, consider the triangle program. At the start, we have the testable input domain, which is just three integers:*



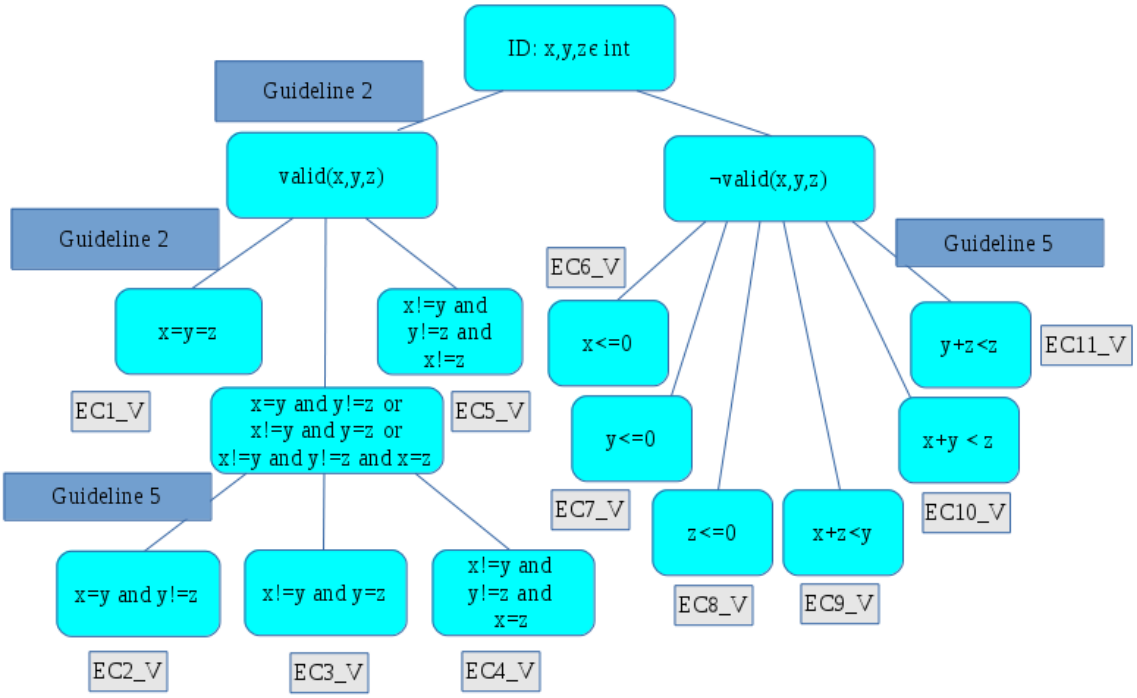
We apply guideline 2 to break this into valid and invalid triangles, giving us two new leaf nodes:



Then, we apply guideline 2 again to the three different types of triangle: equilateral, isosceles, and scalene, breaking the valid triangle node into three new leaf nodes:



Continue the process several more times, we end up with the following test template tree:



Note that the leaf nodes: (a) do not overlap, because we were careful to always break leaf nodes into non-overlapping cases; and (b) cover the entire input space, because we were careful to always ensure that each breakdown covered its parent node.

From here, the 11 equivalence classes fall out directly from the test template tree:

$$
\begin{array}{lcl}
EC1\_V & = & \{x, y, z \in int \mid valid(x, y, z) \wedge x = y = z\} \\
EC2\_V & = & \{x, y, z \in int \mid valid(x, y, z) \wedge x = y \wedge y \neq z\} \\
EC3\_V & = & \{x, y, z \in int \mid valid(x, y, z) \wedge x \neq y \wedge y = z\} \\
EC4\_V & = & \{x, y, z \in int \mid valid(x, y, z) \wedge x \neq y \wedge \wedge y \neq zx = z\} \\
EC5\_V & = & \{x, y, z \in int \mid valid(x, y, z) \wedge x \neq y \wedge y \neq z \wedge x \neq z\} \\
EC6\_V & = & \{x, y, z \in int \mid \neg valid(x, y, z) \wedge x \leq 0\} \\
\ldots & & \\
EC11\_V & = & \{x, y, z \in int \mid \neg valid(x, y, z) \wedge y + z < z\}
\end{array}
$$

Mitigating Tree Explosion

Generating partitions like this can result in trees that are prohibitively large. Even the tree for the triangle example above starts to become tedious, and this is a tiny program. The problem is caused by the nature of the partitioning: if we apply a guideline to every leaf node of a tree, then the number of nodes grows exponentially with the depth of the tree.

To mitigate the problem of explosion, we can apply a couple of rules of thumb:

1. *Test invalid, exceptional, and error cases only once*: Many test inputs involve invalid or exceptional cases, such as throwing an exception or returning an error codes. For these cases, aim to have only one node in the tree that corresponds to each trigger for these exceptional cases. Applying guidelines to exceptional cases will *probably* not (but not certainly not) provide good test inputs. This is because most people use defensive programming, and therefore errors/exceptions are detected at the start of programs and the error/exception is thrown immediately. As such, the remainder of the program is not executed. So, if a node in a test template tree is designed to throw an exception due to an ill-formed input, creating child partitions that test several different inputs for 'normal case' functionality are unlikely to find any additional faults because the

corresponding code will not be executed.

2. *Look at variable interactions*: When applying a guideline, use your experience and intuition and apply only to the current leaf nodes in which this partition is likely to involve some interaction between variables. For example, consider a system for recording loans from a library, with inputs such as date of loan, length of loan, name of book/DVD, author, etc. There is a clear interaction between the date of the loan and the length of the loan, because these two variables determine the due date. However, the name of the book and the author (probably) do not affect the due date. As such, when testing different values of loan length, it would be better to prioritise the nodes that focus on the loan date, rather than on the book name and author. This reduces the number of partitions applied, and therefore mitigates the problem of explosion.

In use

In reality, software engineers/testers do not tend to capture these test template trees in diagrams. The idea of the test template tree is merely as a way to structure one's thinking in a divide-and-conquer approach to testing. The trees can be constructed loosely in one's head to provide a top-down view of testing, or can be sketched when the task is too large to contain mentally.

That said, I have been in projects where these types of models are sketched out on whiteboards for system-level testing to provide an overview of a high-level testing strategy. So, this idea can be used as a communication tool about testing in an industry project in the same way that we are using it in these notes.

2.5 Combining Partitions

The partitioning techniques in this section advocate looking at the variables of a program, and using test case selection techniques to derive equivalence classes. However, we quickly run into problems in which we use multiple criteria, or one criteria over multiple variables.

For example, recall in Example 15, in which we derived test cases for the triangle program using equivalence partitioning, that we generated six overlapping equivalence classes for invalid triangles. This was done by using two different criteria over three different variables. Removing the overlapping cases would generate a large number of test cases, many of which would be of little use. For example, three of these cases would be the following:

$$\mathbf{x} \leq \mathbf{0} \wedge y > 0 \wedge z > 0 \wedge x + y \ldots$$
$$x > 0 \wedge y > 0 \wedge \mathbf{z} \leq \mathbf{0} \wedge x + y \ldots$$
$$\mathbf{x} \leq \mathbf{0} \wedge y > 0 \wedge \mathbf{z} \leq \mathbf{0} \wedge x + y \ldots$$

The bold in these represent the values that make the triangle invalid. However, programs statements are executed in sequence, so an implementation of the triangle problem would check for non-positive x and z, but would check one before the other. This makes the third case redundant.

Redundancy is not the only issue. If we consider using a partitioning method on three different variables, one with the domain of non-negative integers (variable x), one with the domain of non-positive integers (variable y, and one with the domain of characters z, we may end up with the following partitions:

(i) For x: $x = 0$, $0 < x < 10$, and $x \geq 10$ (three equivalence classes).

(ii) For y: $y = 0$ and $y > 0$ (two equivalence classes).

(iii) For z: z is a lower-case letter, z is an upper-case letter, and z is a non-letter character (three equivalence classes).

Ammann and Offutt [1] call these *block* combinations, in that, while the equivalence classes overlap, there are distinct blocks that are different to each other in some way. In the above case, the blocks are based on the variables, and there are equivalence classes for each variables.

When we combine the variables, we encounter overlap. The question is how to resolve the overlap so that we do not derive too many test cases. We present three different criteria to solve the problem: the *all combinations* criterion, the *pair-wise combinations* criterion, and the *each choice combinations* criterion.

All Combinations

The *all combinations* criterion specifies that every combination of each equivalence class between blocks must be used. This is analogous to the *cross product* of sets. So, if we considering the program with the inputs x, y, and z, then we would require 18 test cases ($3 \times 2 \times 3$), satisfying the following 18 equivalence classes, consisting of the following 6 equivalence classes:

$$\begin{array}{l} x = 0 \wedge y = 0 \wedge z \text{ is lowercase} \\ x = 0 \wedge y = 0 \wedge z \text{ is uppercase} \\ x = 0 \wedge y = 0 \wedge z \text{ is a non-letter} \\ x = 0 \wedge y > 0 \wedge z \text{ is lowercase} \\ x = 0 \wedge y > 0 \wedge z \text{ is uppercase} \\ x = 0 \wedge y > 0 \wedge z \text{ is a non-letter} \end{array}$$

and the remaining twelve, which are as above except with $0 < x < 10$ and $x \geq 10$.

The number of test cases is a product of the number of blocks and the number of equivalence classes in each block. In this case, it relates to the number of variables, and the number of equivalence classes for each variable. Therefore, if there was two additional variables, each with three equivalence classes, the number of test cases would be $3 \times 2 \times 3 \times 3 \times 3 = 162$ test cases. This is likely to be more test cases than necessary, as we saw with the triangle program.

Each Choice Combinations

The *each choice* criterion specifies that just one test case must be chosen from each equivalence class. This is the approach we took in the testing of triangle program in Example 15.

Taking the example of the three variable program, the each choice criterion could be satisfied by choosing inputs that satisfy the following three equivalence classes:

$$\begin{array}{l} x = 0 \wedge y = 0 \wedge z \text{ is lowercase} \\ 0 < x < 10 \wedge y > 0 \wedge z \text{ is uppercase} \\ x \geq 10 \wedge y = 0 \wedge z \text{ is a non-letter} \end{array}$$

This is significantly weaker that the all combinations criterion, and does not consider combinations of values.

Pair-Wise Combinations

The *pair-wise combinations* criterion aims to combine values, but not an exhaustive enumeration of all possible combinations. As the name suggests, an equivalence class from each block must be paired with every other equivalence class from all other blocks.

In our above example, this implies that we must have test cases to satisfy the following 15 pair-wise combinations:

$x = 0 \wedge y = 0$	$0 < x < 10 \wedge y = 0$	$x \geq 10 \wedge y = 0$
$x = 0 \wedge y > 0$	$0 < x < 10 \wedge y > 0$	$x \geq 10 \wedge y > 0$
$x = 0 \wedge z$ is lowercase	$0 < x < 10 \wedge z$ is lowercase	$x \geq 10 \wedge z$ is lowercase
$x = 0 \wedge z$ is uppercase	$0 < x < 10 \wedge z$ is uppercase	$x \geq 10 \wedge z$ is uppercase
$x = 0 \wedge z$ is a non-letter	$0 < x < 10 \wedge z$ is a non-letter	$x \geq 10 \wedge z$ is a non-letter

However, a single test case can cover more than one of these. For example, the tester case $x = 0 \wedge y = 0 \wedge z$ is lowercase

covers both $\mathsf{x} = 0 \wedge \mathsf{y} = 0$ and $\mathsf{x} = 0 \wedge \mathsf{z}$ is lowercase. With this in mind, we can select test cases that satisfy only nine equivalence classes; the three classes below:

$$\mathsf{x} = 0 \wedge \mathsf{y} = 0 \wedge \mathsf{z} \text{ is lowercase}$$
$$\mathsf{x} = 0 \wedge \mathsf{y} > 0 \wedge \mathsf{z} \text{ is uppercase}$$
$$\mathsf{x} = 0 \wedge \mathsf{y} > 0 \wedge \mathsf{z} \text{ is a non-letter}$$

and the remaining six, which are as above except with $0 < \mathsf{x} < 10$ and $\mathsf{x} \geq 10$.

One can generalise this to *T-Wise Combinations*, in which we require T number of combinations instead of pairs. It T is equal to the number of blocks, then this is equivalent to the all combinations criterion.

2.6 References

[1] P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008.

[2] R. Hierons, Avoiding coincidental correctness in boundary value analysis, *ACM Transactions on Software Engineering and Methodology*, 15(3):227–241, 2006.

Chapter 3

Boundary-Value Analysis

The input partitioning methods presented in Chapter 2 can be improved by trying to find better test cases within equivalence classes. *Boundary-value analysis* is both a refinement[1] of input partitioning and an extension of it. Boundary-value analysis selects test cases to explore conditions on, and around, the edges of equivalence classes obtained by input partitioning.

> **Definition 18** *Boundary conditions* are predicates that apply directly on, above, and beneath the boundaries of input equivalence classes and output equivalence classes.

Intuitively, boundary-value analysis aims to select test cases to explore the boundary conditions of a program. Boundary-value analysis and input partitioning are closely related. Both of them exploit the idea that each element in an equivalence class should execute the same paths in a program. However, boundary-value analysis works on the theory that, if a programmer makes a mistake in the logic of the program, such that some inputs in an equivalence class execute the incorrect paths, then that mistake is more likely to be at the *boundary* between equivalence classes, because these boundaries are related to flow control constructs such as if statements and while loops in programs.

For example, if we have a specification that states that a program behaves a specific way if the length of a list is greater than 10, and a different way if it is not, then one may expect to see a branch statement similar to the following.

```
if (length(list) > 10) then ...
else ...
```

There are two equivalence classes for this: when the length of list is greater than 10, and when it is not. Selecting test cases arbitrarily can give us two test inputs: $\mathsf{list} = [1\,..\,15]$, and $\mathsf{list} = [1\,..\,5]$. However, these test cases would not detect the following fault.

```
if (length(list) >= 10) then ...
else ...
```

Here, the boundary has shifted across one value. Boundary-value analysis attempts to find these faults near the boundary by selecting test cases on and around the boundary. After identifying the equivalence classes, we would select *at least* test cases that satisfy the condition cases: $\mathsf{length}(\mathsf{list}) = 10$ (the top end of the first EC); and $\mathsf{length}(\mathsf{list}) > 10$ (the bottom end of the second EC).

Collected defect data supports the theory behind boundary-value analysis. Faults are more likely to be introduced at boundary conditions because programmers either: (1) are unsure of the correct boundary for an input condition;

[1] The word "refinement" means that we take the basic method of equivalence partitioning and change it to create a test case selection strategy that has a greater probability of finding more faults in the program.

or (2) have incorrectly tested the boundary. Therefore, test cases that explore the boundary conditions have a more important role that test cases that do not.

Boundary-value analysis requires one *or more* test cases be selected from the edge of the equivalence class or close to the edge of the equivalence class, whereas equivalence partitioning simply requires that any element in the equivalence class will do. Boundary-value analysis also requires that test cases be derived from the output conditions. This is different to equivalence partitioning where only the input domain is usually considered.

3.1 Values and Boundaries

Before we discuss how to apply boundary-value analysis, we define some related terms.

Definition 19

(i) A *path condition* is the condition that must be satisfied by the input data for that path to be executed.

(ii) A *domain* is the set of input data satisfying a path condition.

(iii) A *domain boundary* is the boundary of a domain. It typically corresponds to a simple predicate.

Test inputs are chosen to be *on* or around the boundary. However, there are two distinctly different types of boundary: *closed boundaries* and *open boundaries*.

Definition 20

(i) A *closed boundary* is a domain boundary where the points on the boundary belongs to the domain, and is defined using an operator that contains an equality. For example, $\mathsf{x} =< 10$ and $\mathsf{y} == 0$ are both closed boundaries.

(ii) An *open boundary* is a domain boundary which is not closed, and is defined using a strict inequality. For example, $\mathsf{y} < 10$ is an open boundary, because 10 does not fall withing the boundary.

(iii) An *on point* is a point on the boundary of an equivalence. For example, the value 10 is an on point for the boundary $\mathsf{x} =< 10$. Counter-intuitively, 10 is also the on point for the boundary $\mathsf{y} < 10$. Therefore, for a closed boundary, an on point will be a member of the equivalence class, and for an open boundary, it will not.

(iv) An *off point* is a point just off the boundary of an equivalence class. For example, if x is an integer, the value 11 is the off point for the equivalence class $\mathsf{x} =< 10$. If x is a floating point number, the off point would be 10.001 (assuming that 0.001 is the smallest floating point on the specific architecture).

For the open boundary $\mathsf{y} < 10$, the off point is 9 if y is an integer, and 9.999 is y is a floating point number.

The reverse of the on point then holds: for a closed boundary, the off point will fall outside of the equivalence class, and for an open boundary, it will fall inside the equivalence class.

A special case of off points is strict equalities. That is, boundaries of the form $\mathsf{x} == 10$. In this case, there are two off points: 9 and 11.

3.2 Test Case Selection Guidelines for Boundary-Value Analysis

Given a set of domain boundaries to test, we can apply the following guidelines to select test cases.

1. If a boundary is a strict equality, for example $\mathsf{x} == 10$ select the on point, 10, and both off points, 9 and 11.
2. If a boundary is an inequality, for example $\mathsf{x} < 10$, select the on point 10, and the off point 9.

3. For boundaries containing unordered types, that is, types such as enumerations of strings, choose one on point and one off point. An on point is any value that is in the equivalence class, while an off point is any value that is not. For example, if the equivalence class for a string variable is "contains no spaces", then test one string containing no spaces, and one string containing at least one space.
4. Analyse boundaries of equivalance classes; not individual equivalence classes. If we have the equivalences classes $\mathsf{x} =< 0$ and $\mathsf{x} > 0$, then 0 is the on point for both, so analyse that boundary not each class in isolation.

Remark 21 *It is not always easy to see the boundaries for an equivalence class; boundaries may not exist or make sense. When they do, they give better test cases than equivalence partitioning.*

Example 22 *Boundary-Value Analysis for the Triangle Program*

Consider again the triangle classification program from Example 15. To show the boundary-value analysis method more clearly we will change the specification slightly to input floating point numbers instead of integers.

> The program reads floating point values from the standard input. The three values are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle, if it can be formed, is equilateral, isosceles, scalene, or invalid.

Given the equivalence classes identified above using equivalence partitioning we now have the following possible boundary conditions. Let x, y and z be the three floating point numbers input to the program.

1. For an *equilateral* triangle the sides must all be of equal length and we have only one boundary where $x = y = z$ and we can explore this boundary using the following test cases:

   $(3,\ 3,\ 3)$ (on point)
   $(2.999,\ 3,\ 3)$ (off point below)
   $(3.001,\ 3,\ 3)$ (off point above)

   Plus six additional cases for the off points of y and z.
2. For an *isosceles* triangle two sides must be equal. This gives the boundary conditions $x = y \wedge y \neq z$, $y = z \wedge z \neq x$ or $x = z \wedge z \neq y$. The boundary $x = y \wedge y \neq z$ can be explored using the following test cases:

   $(3,\ 3,\ 4)$ (on point)
   $(2.999,\ 3,\ 4)$ (off point below)
   $(3.001,\ 3,\ 4)$ (off point above)

   Plus six additional cases for the off points of y and z.
3. For a *scalene* triangle, all sides must be of a different length. This equivalence class is $x \neq y \wedge y \neq z \wedge z \neq x$. This can be explored using the following test cases:

   $(3,\ 3,\ 3)$ (on point)
   $(2.999,\ 3,\ 3)$ (off point below)
   $(3.001,\ 3,\ 3)$ (off point above)

   Plus six additional cases for the off points of y and z. However, note that all nine of these values have already been tested. The on and off points for this are the same as for the equilateral triangle.
4. For invalid triangles, we have the cases in which the sides are of size 0 or below; that is $x \leq 0$. The following test cases explore this for x:

   $(0,\ 3,\ 3)$ (on point)
   $(0.001,\ 3,\ 3)$ (off point)

   and similarly for y and z.

   For invalid triangles, we also have the cases in which one of the sides is at least as long as the sum of the other two. The following test cases explore the equivalence class $\mathsf{x} + \mathsf{y} \leq \mathsf{z}$:

   $(1,\ 2,\ 3)$ (on point)
   $(1,\ 2,\ 2.999)$ (off point)

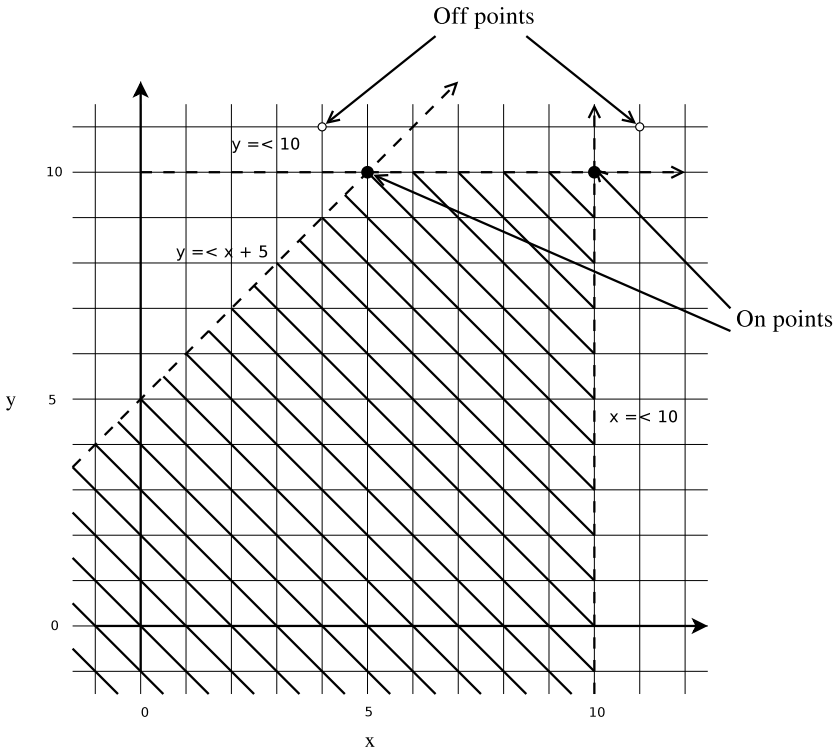


Figure 3.1: On points and off points in a two-dimensional, linear boundary

3.2.1 Domains with Multiple Variables

So far, we have discussed strategies for choosing on and off points for domains containing only single boundaries. However, it is common for domains to have more than one variable. In fact, the triangle program in Example 22 does have equivalence classes with multiple variables, (for example, the invalid case of $x + y > z$), however, we conveniently did not discuss this, because the boundaries were straightforward to identify. In other cases, the boundary is not so clear cut.

First, let us consider test cases for two-dimensional linear boundaries. Consider an equivalence class made up of the following linear boundaries: $x \leq 10$, $y \leq 10$, and $y \leq x + 5$, in which x and y are integers. This can be visualised using the two-dimensional graph in Figure 3.1, with the shaded section identifying the equivalence class.

In this case, we have to choose on point values that satisfy all three of the linear boundaries. However, the intersections of the lines generally make the best test cases, because they test more than one on point in a single test case. This is advantageous because it reduces the number of test cases, and gives us test cases that are more sensitive to boundary-related faults. In the above, we choose two on points: $x = 10 \wedge y = 10$, and $x = 5 \wedge y = 10$.

Similarly, the off points near intersections make the best test cases outside the equivalence class. In Figure 3.1, we choose the test inputs $x = 11 \wedge y = 11$, and $x = 4 \wedge y = 11$. The latter is an off point for $y \leq 10$ as well as $y \leq x + 5$.

Example 23 *The Triangle Program — Again!*

If we return to the triangle example again, we see that there are equivalence classes based on three variables. Therefore, we need to consider an three-dimensional boundary. This is harder to visualise (and anything above three dimensions even more so!), but selecting the on and off points is still straightforward for this particular program.

If we consider the invalid case in which $x + y \leq z$ (that is, the length of the side z is greater than the sum of the other two sides), then we need to select an on and an off point for this. Similar cases exist for when x and y are too long. The inequality, $x + y < z$, describes a *plane* in a three-dimensional graph, rather than a line in a two-dimensional graph. To choose test inputs, we must choose one on point of the plane, and one off point.

If we return to the test cases that we derived for these in Example 22, one can see that the test cases selected were on and off points for this plane:

$(1,\ 2,\ 3)$	(on point)
$(1,\ 2,\ 2.999)$	(off point)

The point $(1, 2, 3)$ is on the plane $x + y \leq z$, while the point $(1, 2, 2.999)$ is just off it. However, is this enough to detected a boundary shift?

Detecting Boundary Shifts in Inequalities

Figure 3.2 shows a boundary with an on and an off point. The solid line represents the boundary that has been identified (for equivalence partitioning, this is the expected boundary derived from the functional requirements, while for domain testing, this is the actual boundary in the program), while the dashed line represents a shifted boundary (for equivalence partitioning, this is the boundary in the program, while for domain testing, this is the boundary derived from the functional requirements). The black dot represents the on point of the boundary, and the white dot the off point.

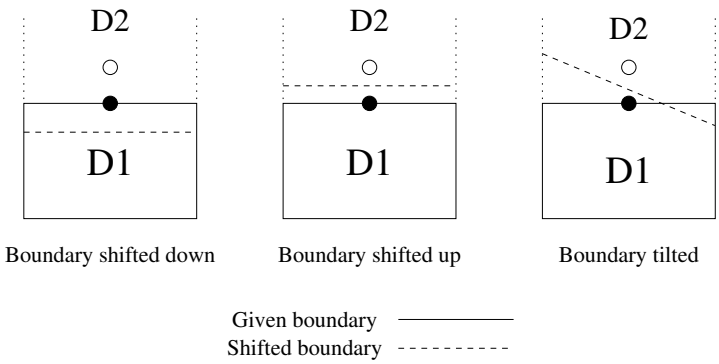


Figure 3.2: Possible inequality boundaries shifts with on and off points

From this figure, one can see that a boundary shift down — in other words, the equivalence class is smaller than expected — will be detected because the on point is identified as being in the wrong domain. However, for the other two, it is possible to have a boundary shift that is not identified.

In the case of a shift up (the equivalence class is larger than expected), the shift has to be great enough that the off point can detect it. For example, if the boundary $x \leq 10.0$ is identified from the functional requirements, then we would select the on point 10.0, and the off point 10.001. However, if this boundary is incorrectly implemented as $x \leq 10.0000000001$ (highly unlikely I'm sure, but this is for illustrative purposes only), then the on point 10.0 still falls inside the boundary, and the off point 10.001 still falls outside the boundary. Therefore, these values will not detect the shift. This means that if the off point is a certain distance away from the boundary, then only shifts at least as big as this distance can be detected by that off point. Fortunately, as Beizer points out in his book *Black-Box Testing — Techniques for Functional Testing of Software and Systems*, faults in programs generally occur naturally, and are not the result of sabotage, so choosing an off point that is close to the boundary is almost always likely to detect boundary shifts.

In the case of the tilted boundary, the given boundary and the shifted boundary intersect at some point. In the diagram above, the on point falls within the boundary, and the off point falls outside of the boundary, therefore, these cases will not detected the boundary shift.

Looking at Figure 3.2, it is clear that select one on point and one off point is not sufficient to detect shifts. For the

middle case, the best that can be done is to select the off point as close to the boundary as possible, which may be impossible in some domains.

However, we can improve on the case of the tilted boundary in some cases. If the values that can be selected have a minimum and a maximum (the *extremum points*), then we choose both of those as on points. This guarantees that, for any tilted boundary, at least one of the points must lie outside the shift boundary. Figure 3.3 demonstrates this. The position of the tilted boundary does not have an impact on this — as long as some of the boundary falls below and some above, then one of the on points will fall outside the domain. If the two on points are close together, then this reduces the chance of the boundary shift being detected.

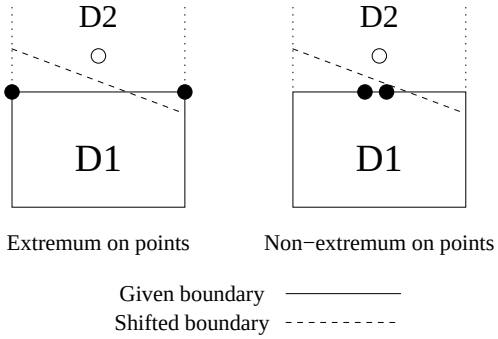


Figure 3.3: Tilted inequality boundary with extremum and non-extremum on points

In the case that the boundary has no extremums (or only a minimum or maximum but not both) due to it being infinite, then the further away the two on points are from each other, the greater chance there is of producing a failure.

One the two on points are selected, the best off point to select is one that is as close to the middle as possible between the two on points. This is known as the centroid. Alternatively, one could select more than one off point, but this would likely have little value.

If we return to Figure 3.1, this implies that we are required to add both point on the line $y =< x + 5$, but it must be at a point where $y < 10$ to remain inside the boundary.

Detecting Boundary Shifts in Equalities

For boundaries defined by equalities, the technique is slightly different. This is because the domain is defined by a line (in two dimensional cases) or a plane (in three dimensional cases), rather than a block. Therefore, if we consider the tilted boundary from Figure 3.2, the on point, represented by the black dot, would not fall inside the boundary, because the entire equivalence class is given by the line, rather than the area below the line.

Recall from Section 3.1, that for equalities involving one variable, we select the on point, and both off points (above and below). For a boundary shift up or down, these are enough to detect the shifts larger than the distance between the on point and off points. However, if the on point chosen happens to intersect with the given and shifted boundary, as in Figure 3.4, then the tilted shift will not be detected.

This is similar to the problem with shifted boundaries for inequalities, and as with that problem, choosing two on points removes this. However, unlike inequality boundaries, equality boundaries are strictly equal, therefore choosing extremum on points is unnecessary. If we consider the boundary tilt in Figure 3.4, then *any* on point other than the point at the intersection will detect the boundary shift. Therefore, if we have *any* two on points, then if the given and shifted boundaries intersect, by definition, at most one of them can be at this intersection. From this, we conclude that we need to chose two arbitrary points.

The same argument goes for non-equalities — that is, defined by predicates such as $x \neq y + 10$ — except that 10 would be the roles of off points and on points are reversed.

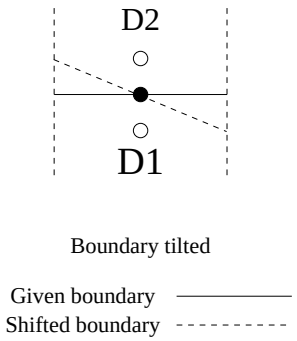


Figure 3.4: Tilted equality boundary

From the above discussion, we conclude the following about selecting boundary values for two-dimensional boundaries:

- For boundaries defined by an inequality, select two on points that are as far apart as possible (or reasonable for infinite domains), and one off point as close to the middle of the two on points as possible;
- For boundaries defined by an equality, select two on points, and two off points (one below and one above); similarly for non-equalities; and
- If the off point is of distance d, then only boundary shifts of magnitude greater than d can be detected.

We can generalise this idea to N dimensional input spaces.

- N on points and 1 off point are needed for boundaries based on inequalities, with the on points as far apart as possible;
- N on points and 2 off point are are needed boundaries based on equalities and non-equalities.
- If the off point test case is a distance d from the boundary, then only boundary shifts of magnitude greater d can be detected.

Chapter 4

Coverage-Based Testing

In this chapter, we discuss three software testing techniques that are based on achieving *coverage* of the software being tested. The notion behind of these techniques is to achieve some form of coverage of the program-under-test, based on well-defined criteria, rather than partitioning the input domain of the program.

There are three techniques that we discuss in this chapter:

1. *control-flow testing* – control-flow strategies select test inputs to *exercise* paths in the control-flow graph. They select paths based on the control information in the graph, typically given by predicates in if statements and while loops.

   Test inputs are selected to meet criteria for *covering* the graph (and therefore the code) with test cases in various ways. Examples of coverage criteria include:

   (i) Path coverage;
   (ii) Branch coverage;
   (iii) Condition coverage; and
   (iv) Statement coverage.

2. *data-flow testing* — data-flow strategies select inputs to exercise paths based on the flow of data between variables; for example, between the *definition* of a variable, such as assigning a value to a variable, and the *use* of that variable in the program

   As with control-flow strategies, test inputs are selected to cover the graph (and therefore the code) with test cases. Examples of coverage criteria include:

   (i) Execute every definition;
   (ii) Execute every use;
   (iii) Execute every path between every definition and every use.

3. *mutation analysis* — mutation analysis is a technique for measuring the effectiveness of test suites, with the side effect that test cases are created and added to that test suite. The technique is based on seeding faults in a program, and then assessing whether or not that fault is detected by the test suite. If not, then that test suite is *inadequate* because the fault was not detected, and a new test case must be added to find that fault.

   Using specially designed operators, many copies of the program-under-testare created, each one with a fault, with the aim of the test suite killing all of these.

   Remark 24 Not that these techniques are useful only for selecting test *inputs*; not test *cases*. That is, they are not useful for selecting test outputs. If both the test inputs and expected outputs are derived from the program itself, then we will never produce failures because the expected outputs and actual outputs will always be the same. As such, a white-box testing technique still requires a test oracle, and therefore, a specification of the program behaviour.

4.1 Control-Flow Testing

In this section, we discuss *control-flow testing* techniques.

4.1.1 Control-Flow Graphs

The most common form of white-box testing is control-flow testing, which aims to understand the *flow of control* within a program. There are a number of ways of capturing the flow of control in a program, but by far the most common is by using a *control-flow graph* (CFG).

Informally, a CFG is a graphical representation of the control structure of a program and the possible *paths* along which the program may execute. More formally we have the following definition.

Definition 25 A *control-flow graph* is a graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ where:

A. A vertex in the CFG represents a program statement;

B. An edge in the CFG represents the ability of a program to flow from its current statement to the statement at the other end of the edge;

C. If an edge is associated with a conditional statement, label the edge with the conditionals value, either true or false.

In diagrams, statements are usually represented by square boxes and branches by diamond boxes.

```
int power(int base, int n)
{
     int power, i;
     power = 1;
     for (i = 1; i <= n; i ++) {
         power = base * power;
     }
     return power;
}
```

Figure 4.1: A C function to iteratively raise a base to a power.

As a first example consider the power function in Figure 4.1, which for integer inputs base and n, calculates $\mathsf{base}^{\mathsf{n}}$. All looping constructs decompose into *test-and-branch* control flow structures. The for loop in Figure 4.1 needs to be decomposed into a test i <= n and a branch. The control graph for the power function appears in Figure 4.2.

We will need the following definitions for our analysis for control flow graphs.

Definition 26

(i) An *execution path*, or just a *path*, is a sequence of nodes in the control flow graph that starts at the entry node and ends at the exit node.

(ii) A *branch*, or *decision*, is a point in the program where the flow of control can diverge. For example, if-then-else statements and switch statements cause branches in the control flow graph.

(iii) A *condition* is a simple *atomic* predicate or simple relational expression occurring within a branch. Conditions *do not* contain *and* (in C &&), *or* (in C ||) and *not* (in C !) operators.

(iv) A *feasible path* is a path where there is at least one input in the input domain that can force the program to execute the path. Otherwise the path is an *infeasible path* and no test case can force the program to execute that path.

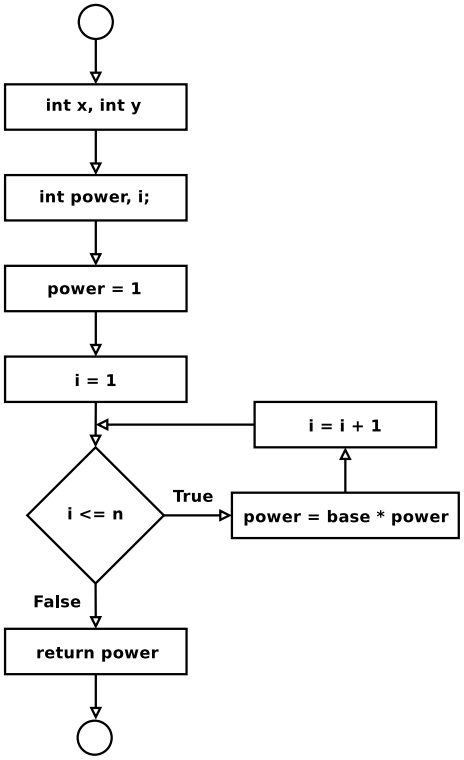
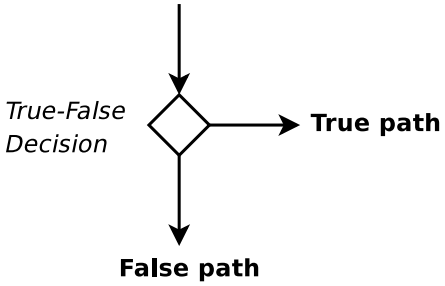


Figure 4.2: The control-flow graph for the function in Figure 4.1

In terms of the control-flow graph, a branch is a node in the graph with two or more edges that leave that node:



In turn branches are made up from conditions. For example, the branch given by

$$\text{if } ((\text{a} > 1) \,\&\&\, (\text{b} == 0))$$

consists of the conjunction of two conditions (a > 1) and (b == 0). Analysing the branches and conditions in a program gives us a great deal of insight into how to choose test cases to follow specific paths, for example, we need to select test inputs to make a branch true and false and in turn this means choosing values for a and b to make the branch take on the vales true and false.

In this example, there is one way to make the branch true:

$$((\text{a} > 1) \,\&\&\, (\text{b} == 0)),$$

and three ways to make the branch false

$$((\text{a} > 1) \,\&\&\, (\text{b}! = 0)), ((\text{a} =< 1) \,\&\&\, (\text{b} == 0)), \text{and} ((\text{a} =< 1) \,\&\&\, (\text{b}! = 0))$$

4.1.2 Coverage-Based Criteria

The aim of coverage-based testing methods is to *cover* the program with test cases that satisfy some fixed coverage criteria. Put another way, we choose test cases to exercise as much of the program as possible according to some criteria. If some part of the program is not exercised by any test case then there may well be undiscovered faults lurking there.

Coverage-based testing for control-flow graphs works by choosing test cases according to well defined *coverage* criteria. The more common coverage criteria are the following.

- *Statement coverage* (or *node coverage*)
  Every statement of the program should be exercised at least once.
- *Branch coverage* (or *decision coverage*)
  Every possible alternative in a branch (or decision) of the program should be exercised at least once. For if statements this means that the branch must be made to take on the values true and false, *even if the result of this branch does nothing*, for example, for an if statement with no else, the false case must still be executed.
- *Condition coverage*
  Each condition in a branch is made to evaluate to true and false at least once. For example, in the branch if(a&&b), a must evaluate to true and false at least once, and so must b.
- *Decision/Condition coverage*
  Each condition in a branch is made to evaluate to both true and false and each branch is made to evaluate to both true and false. That is, a combination of branch coverage and condition coverage. For example, in the branch if(a&&b), a must evaluate at least a == true/false, b == true/false, a&&b == true/false. This can be achieved using the following:

a	b	a && b
true	true	true
false	false	false

- *Multiple-condition coverage*
  All possible combinations of condition outcomes within each branch should be exercised at least once. For example, in the branch if(a&&b), a must evaluate all combinations of a and b:

a	b	a && b
true	true	true
false	true	false
true	false	false
false	false	false

- *Path coverage*
  Every execution *path* of the program should be exercised at least once.

Note that path coverage is impossible for graphs containing loops, because there are an infinite number of paths. A typical work around is to apply the *zero-to-many* rule, discussed in Section 2.2, which states that the minimum number of times a loop can execute is zero, so this is a boundary condition. Similarly, some loops have an upper bound, so treat this as a boundary. A general guideline for loops is to select test inputs that execute the loop:

- zero times – so that we can test paths that do not execute the loop;
- once – to test that the loop can be entered and that the results for a single iteration are "correct";
- twice – to test that the results remain "correct" between different iterations of the loop;
- N times (greater than 2) – to test that an arbitrary number of iterations returns the "correct" results; and
- N+1 times to test that after an arbitrary number of iterations the results remain "correct" between iterations.

In the case that we know the upper bound of the loop, then set N+1 to be this upper bound.

Example 27 *A small example*

To motivate the selection of test cases consider the following simple program in Figure 4.3.

```
void main(void)
{
    int a, b, c;
    scanf("%d %d %d", a, b, c);

    if ((a > 1) && (b == 0)) {
        c = c / a;
    }

    if ((a == 2) || (c > 1)) {
        c = c + 1;
    }

    while (a >= 2) {
        a = a - 2;
    }

    printf("%d %d %d", a, b, c);
}
```

Figure 4.3: A simple program for control-flow testing.

The first step in the analysis is to generate the control-flow graph which we show in Figure 4.4.

Now, what is needed for statement coverage? If all of the branches are true at least once then we will have executed every statement in the graph. Put another way to execute every statement at least once we must execute the path ABCDEFGF.

Now, looking at the *conditions* inside each of the three *branches* we can derive a set of constraints on the values of a, b and c such that a>1 and b $==$ 0 in order to make the first branch true, a $\leq$ 2 in order to make the third branch true, and c>1 to make the second branch true. A test case of the form (a, b, c) $=$ (2, 0, 3) will execute all of the statements in the program.

Note that we have not needed to make every branch take on both values true and false, nor have we made every condition evaluate to true and false, nor have we traversed every path in the program.

We already have a test input (2, 0, 3) that will make all of the branches true. To meet the branch coverage condition all we need are test inputs that will make each branch false. Again looking at the conditions inside the branches, the test input (1, 1, 1) will make all of the branches false. So our two test inputs (2, 0, 3) and (1, 1, 1) is a test suite that will meet the branch coverage criteria.

For any of the criteria involving condition coverage we need to look at each of the five conditions in the program: $C_1 =$ (a>1), $C_2 =$ (b $==$ 0), $C_3 =$ (a $==$ 2), $C_4 =$ (c>1) and $C_5 =$ (a $>=$ 2). The test input (1, 0, 3) will make C_1 false, C_2 true, C_3 false, C_4 true and C_5 false. Examples of sets of test inputs and the criteria that they meet are given in Table 4.1.

The set of test cases meeting the multiple condition criteria is given in Table 4.2. In the table we let the branches $B_1 = C_1 \&\& C_2$, $B_2 = C_3 \,||\, C_4$ and $B_3 = C_5$.

Exercise 1

(i) Can you find an infeasible path *in this example?*

Reminder*: A path is said to be* infeasible *if there are no test cases in the input domain that can execute the path.*

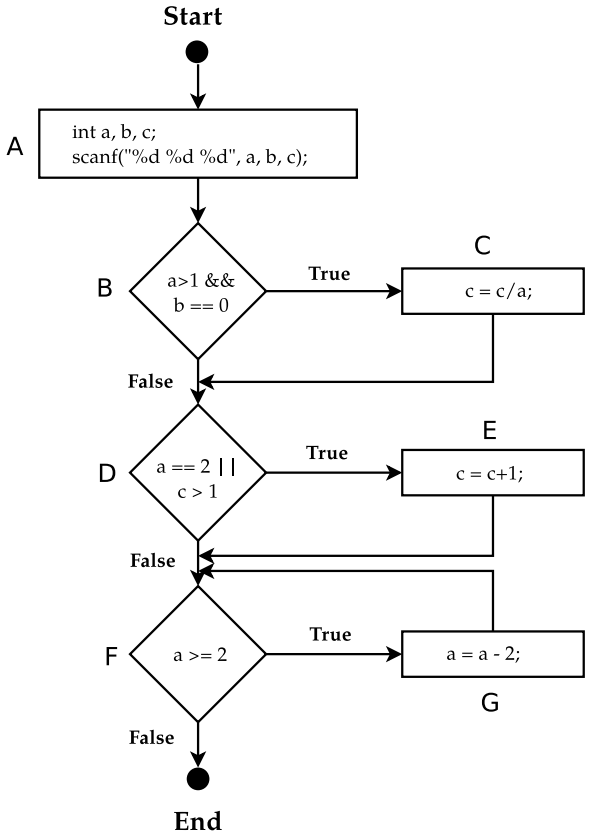


Figure 4.4: The Control Flow Graph for the program in Figure 4.3

Coverage Criteria	Test Inputs (a, b, c)	Execution Paths
Statement	(2, 0, 3)	ABCDEFGF
Branch	(2, 0, 3),	ABCDEFGF
	(1, 1, 1)	ABDF
Condition	(1, 0, 3),	ABDEF
	(2, 1, 1)	ABDFGF
Decision/ Condition	(2, 0, 4),	ABCDEFGF
	(1, 1, 1)	ABDF
Multiple Condition	(2, 0, 4),	ABCDEFGF
	(2, 1, 1),	ABDEFGF
	(1, 0, 2),	ABDEF
	(1, 1, 1)	ABDF
Path	(2, 0, 4),	ABCDEFGF
	(2, 1, 1),	ABDFGF
	(1, 0, 2),	ABDEF
	(4, 0, 0),	ABCDFGFGF
	...	...

Table 4.1: Test inputs for the various coverage criteria for the program in Figure 4.3

			B_1			B_2	B_3
Test	C_1	C_2		C_3	C_4		C_5
inputs	a > 1	b==0		a==2	c > 1		a ≥ 2
(1,0,3)	F	T	F	F	T	T	F
(2,1,1)	T	F	F	T	F	F	T
(2,0,4)	T	T	T	T	T	T	T
(1,1,1)	F	F	F	F	F	F	F

Table 4.2: Multiple condition coverage for the program in Figure 4.3

(ii) How many execution paths are there in the program?

Example 28 *A Second Example*

For the second example, consider the function squeeze from Section 1.2 (shown again in Figure 4.5).

```
void squeeze(char s[], int c)
{
    int i,j;
    for (i = j = 0; s[i] != '\0'; i++) {
        if (s[i] != c) {
            s[j++] = s[i];
        }
    }
    s[j] = '\0';
}
```

Figure 4.5: The squeeze function from Kernighan and Ritchie revisited.

Our analysis begins with the standard analysis of input/output domains and then constructs a control-flow graph for the function. The input domain for the function squeeze is char[] × int and the output domain is char[]. The control-flow graph is shown in Figure 4.6.

The next step is to decide which of the coverage criteria will be used to select test cases. Typically, the exact criteria (statement, branch, condition, multiple condition, path) will depend on the project and what testing is meant to achieve.

For this example we will begin by determining the branches and conditions. Fortunately there are only two branches each of which contains only a single condition. Consequently, if we choose test inputs so that each condition evaluates to both true and false then we will also have achieved branch coverage.

Notice that the paths divide the input domain into subsets where each element of a subsets selects a specific path. For the squeeze functions the subsets can be characterised as follows.

Path	**Test Inputs to Select the Path**
ABCG	Selected by the test input (“”, c) for any character c.
ABCDFCG	Selected by any test input (S, c) where S is a string of length 1 and c occurs in S.
ABCDEFCG	Selected by any test input (S, c) where S is a string of length 1 and c does not occur in S.
⋮	
ABCDFCDF...CDFCG	Selected by any test input (S, c) where S is a string of length n and c does not occur in S.
ABCDEFCDEF...CDEFCG	Selected by any test input (S, c) where S is a string of length n and c occurs in S.

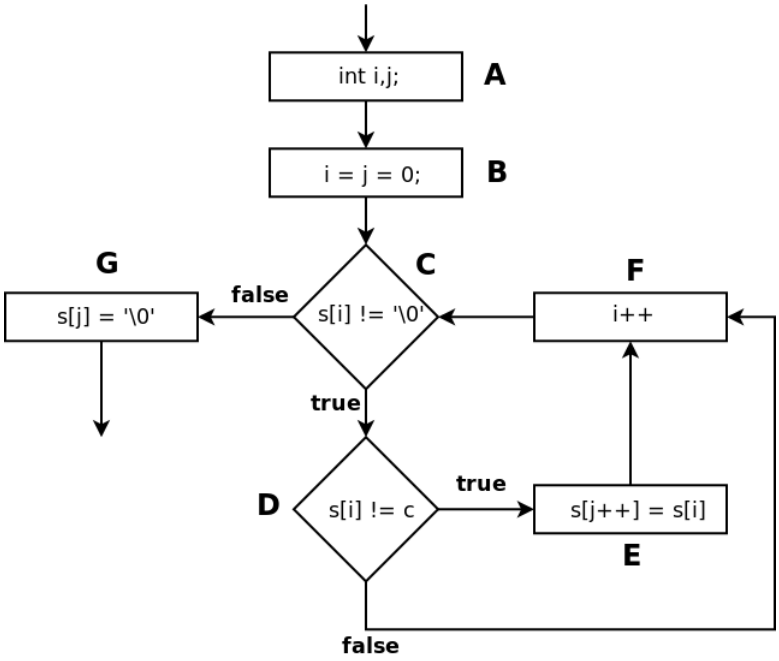


Figure 4.6: The control-flow graph for the squeeze function.

Next we need to derive the test cases. To do this there are two further questions that we need to answer.

(i) How many times do we need to execute the for loop?

(ii) How do we determine the expected results for a test case?

If path coverage were to be demanded, then the coverage criteria could not be satisfied because there are an infinite number of paths. However, we can apply the guideline regarding loops and select test inputs that execute the loop 0, 1, 2, N, and N+1 times. In the squeeze example, the control-flow graph has two loops. Both start and end at node C, however, only one of them contains the node E. Therefore, the zero-to-many rule is applied to each of these. This gives us the paths $ABC(DFC)^nG$ and $ABC(DEFC)^nG$, for $n \in \{0, 1, 2, 4, 5\}$.

This example shows the weakness with the zero-to-many rule (and in path coverage with loops). The paths generated for this only execute the case in which the character c is in every position of the array, and the case in which it is in none. Instead of following this strictly, we use our intuition to combine the two cases, and interleave the loop paths; therefore, in some iterations the test case executes the statement at node E, and in others, it does not.

Now that the test inputs have been derived, the expected outputs must be determined. In our case we will use the design-level specification for the squeeze function to manually determine the expected outputs.

Table 4.3 outlines the test cases, and the paths that they exercise, for the squeeze program.

4.1.3 Measuring Coverage

Clearly, applying control-flow testing manually is a time-consuming process that is unlikely to give many benefits. However, there are aspects of the process that are easily automatable due to their mechanical nature. Deriving the control-flow graph can be automated by simply looking at the source code. Deriving test inputs from is considerably more difficult, especially for programs with non-linear domains, and for non-primitive data types.

However, one straightforward and valuable task is to measure the level of coverage of a given test suite, which can

Coverage Criteria	Test Input	Expected Output	Path
Statement Coverage	(“c”, ‘c’)	“”	ABCDEFCG
Branch Coverage	(“”, ‘c’)	“”	ABCG
	(“ab”, ‘b’)	“a’	ABCDFCDEFCG
Path Coverage	(“”, ‘c’)	“”	ABCG
	(“c”, ‘c’)	“”	ABCDEFCG
	(“ab”, ‘b’)	“a”	ABCDFCDEFCG
	(“abcd”, ‘d’)	“abc”	ABC(DFC)^{3}DEFCG
	(“abcde”, ‘e’)	“abcd”	ABC(DFC)^{4}DEFCG

Table 4.3: Test Cases for the squeeze function.

be automated easily with tools. This can be measured by executing the test suite over the program, seeing which coverage objectives are met, and providing both high-level and fine-grained feedback on the coverage.

A *coverage score* is defined as the number of test objects met divided by the number of total test objectives. The test objectives measured are relative to the particular criterion. For example, for statement coverage, the number of objectives is the number of statements, and the number of objectives met is the number of statements executed by *at least one test*. For branch coverage, the number of objects is the number of branches $\times$ two (each branch must be executed for both the true and false case).

As an example, consider the squeeze function from Figure 4.6. If we want to achieve branch coverage, there are four test objectives: two branches, each with two possible outputs. Now, consider the following test suite, consisting of two tests, which could be generated using any arbitrary technique: `(s = ‘‘abc’’, c = ‘‘d’’)`, `(s = ‘‘’’, c = ‘‘d’’)`.

We can measure the coverage of this as follows, in which e.g. the FFFT in the first row indicates that branch C was executed *true* on the first three iterations of the loop, and *false* on the final iteration:

Inputs	`s[i] != ‘\0’`	`s[i] != c`
`(s = ‘‘abc’’, c = ‘‘d’’)`	TTTF	FFF
`(s = ‘‘’’, c = ‘‘d’’)`	TTTF	FFF

Scanning down the columns, we can see that the branch `s[i] != ‘\0’` is executed for both true and false at least once, while the branch `s[i] != c` is executed only for the false case. The coverage score is calculated as:

$$\begin{aligned} & \text{objectives met} \div \text{total objectives} \\ = \; & \tfrac{3}{4} \\ = \; & 75\% \end{aligned}$$

This means that we have achieved 75% branch coverage. Further, looking at the table, we can scan down the columns to see *which* coverage objectives were not met (the true case for `s[i] != c`), and can add a new test input to cover this case.

Thus, from a practical perspective, the idea is to generate a test suite using some method (generally a black-box method), and to then measure the coverage of that test suite. Test objectives that are not covered can then be added using the technique described in this section. Such an approach is cheaper than deriving tests from the control-flow graph directly, due to the fact that a reasonable set of black-box tests will generally achieve a high coverage initially.

4.1.4 Tool Support

There are many control-flow coverage tools available for most languages, although many of these are restricted only the statement coverage, and sometimes branch or decision/condition coverage. Some open-source tools for Java measuring Java code coverage can be found at `http://java-source.net/open-source/code-coverage`.

One excellent piece of software support is Atlassian's Clover tool (`https://www.atlassian.com/software/clover/overview`). Clover provides feedback on statement (node), branch, and method coverage, and reports fine-grained details such as which tests cover which parts of the code.

4.2 Data-Flow Testing

It is hard to do better to motivate data-flow testing than that given by Rapps and Wuyeker:

> *It is our belief that, just as one would not feel confident about a program without executing every statement in it as part of some test, one should not feel confident about a program without having seen the effect of using the value produced by each and every computation.*

Despite the analyses of input and output domains used in equivalence partitioning, boundary-value analysis, and control-flow graphs, testing still relies on personal experience to choose test cases that will uncover program faults. The problem is made harder if a programs has a large number of input variables. For example, if a program with 5 input variables and each variable's input domain is partitioned into 5 equivalence classes then there are $5^5 = 3125$ possible combinations to test.

How can we select better test cases in order to increase the power of each test case?

One way of improving our test cases is to find better criteria for test case selection. The other way is to find a testing a scheme that produces additional information (information other than the output of the program under test) to use for program analysis.

Data-flow analysis provides information about the creation and use of data definitions in a program. The information generated by data-flow analysis can be used to:

- detect many simple programming or logical faults in the program;
- provide testers with dependencies between the definition and use of variables;
- provide a set of criteria to complement coverage based testing.

Static analysis of programs is a large field. We will go into enough depth in these notes to give you the general idea, some techniques that you can use in practice and hopefully to extend later for specific applications.

4.2.1 Static Data-Flow Analysis

In the simplest form of data-flow analysis, a programming language statement may act on a variable in 3 different ways. It may *define* a variable, *reference* a variable and *undefine* a variable.

Define (d) – A statement *define*s a variable by assigning a value to the variable. For example if the program variable x has been declared then the statements:

$$\mathsf{x} = 5;$$

and

$$\mathsf{scan}(\mathsf{x});$$

both defined the variable x, but the statement

$$x = 3 * y;$$

only defines x if y is declared and defined.

Reference (r) – A statement makes a *reference* to a variable either as an:

(1) *l-value* — that is any variable, or array cell, or pointer into which values can be stored. For example, a variable that appears on the left hand side of an assignment statement; or as an

(2) *r-value* — that is any variable that must be referenced to get its value. For example, a variable that appears on the right hand side of an assignment statement.

If the program variable x is both declared and defined then an example of x used as an l-value is the statement

$$x = 5;$$

where it is used to store the value 5. A use of x as an r-value is in the statement

$$y = 3 * x;$$

where x is used to store a value and must be referenced to obtain its current value.

Undefine (u) – A statement *undefine*s a variable whenever the value of the variable becomes unknown. For example, the scope of a local variable ends.

The aim of data-flow analysis is to trace through the program's control-flow graph and detect *data-flow anomalies*. data-flow anomalies indicate the possibility of program faults. The following is a list of the data-flow anomalies.

- A *u-r* anomaly occurs when an undefined variable is referenced. Most commonly *u-r* anomalies occur when a variable is referenced without it having been assigned a value first — that is, it is *uninitialised*. A common source of *u-r* anomalies arises when the wrong variable is referenced.
- A *d-u* anomaly occurs when a defined variable has not been referenced before it becomes undefined. This anomaly usually indicates that the wrong variable has been defined or undefined.
- A *d-d* anomaly indicates that the same variable is defined twice causing a hole in the scope of the first definition of the variable. This anomaly usually occurs because of misspelling or because variables have been imported from another module.

Example 29 *Static Data-Flow Anomalies*

As a first example consider the program fragment in Figure 4.7. Perhaps the most obvious indication of a fault in the program is a *u-r* anomaly where we reference the variable p2 which is undefined at the point where its value is needed. However, in large systems this kind of anomaly can be difficult to detect manually because of branching in the control-flow graph.

The block of program between the definition of p2 and the deletion of p2 is not harmful. However, the anomaly can indicate the premature deletion of p2 which may effect the execution of the program in the region after the deletion of p2.

The second example of a *u-r* anomaly is the function Faulty_Fibonacci given in Figure 4.8. The control-flow graph for Faulty_Fibonacci is given in Figure 4.9.

We need to make some observations about the control-flow graph before going on the have a look at the data-flow issues.

Remark 30

(i) Firstly note that we have divided the CFG into a number of *basic blocks*. Each basic block is a sequence of statements with no branches and only a single entry point and a single exit point and so, only a single path through the block.

The use of basic blocks makes large CFGs easier to visualise and analyse.

```
/* Defining  p1 and p2 */

TestClass1 p1 = new TestClass1();
TestClass2 p2 = new TestClass2();

/* References to p1 and p2 */
p1->process();
p2->process();

/* Reference to p1 */
p1->process2();

delete p2;
/* p2 is now out of scope and so undefined */

/* Reference to p2 which is now undefined */
p2->process2();
...
```

Figure 4.7: A program module with a u-r anomaly.

```
int Faulty_Fibonacci(int n)
{
    int i, j, sum;
    int *c;

    c = malloc(sizeof(double) * (n + 1));
    if(c == NULL) {
        FatalError(OUT_OF_SPACE);
    }

    c[0] = 1;
    c[1] = 1;
    sum = 2;

    for(i = 2; i <= n; i++) {
        c[i] = c[i - 1] + c[i - 2];
        sum += c[i];
        free(c);
    }
    return sum;
}
```

Figure 4.8: A second example of a u-r anomaly.

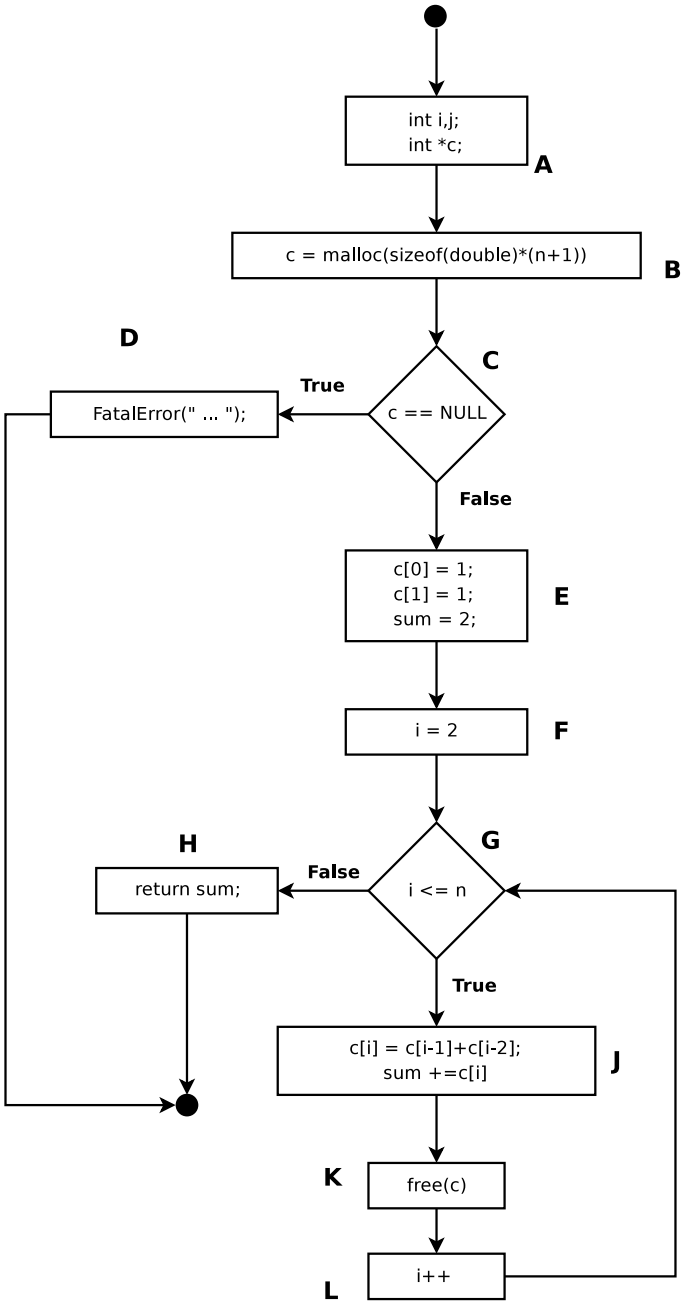


Figure 4.9: The control-flow graph for the program fragment in Figure 4.8, which contains a u-r anomaly.

(ii) Secondly, we have labelled all of the nodes in the graph with letters. In general, since paths are sequences of nodes in the control-flow graph we will label nodes and not edges in the graph.

The fault in the Faulty_Fibonacci program is that we have freed up our memory, the array C in this case, too early. *Our data-flow analysis will only pick this up if the loop is executed two or more times*.

If we look firstly at the path ABCEFGJKL then the variable c undergoes the following sequence of transitions at each of the nodes on the path above.

Node	**Action on the Variable** c
A	no action
B	define (d)
C	reference (r)
E	define (d)
	define (d)
F	no action
G	no action
J	reference (r)
	reference (r)
	define (d)
K	undefine (u)
L	no action

If we execute the loop again we must now revisit nodes G and J. The next action to take on node J is a reference (r) action and so we have a u-r anomaly from the first iteration of loop. This kind of scenario is quite typical of u-r anomalies in C.

As our third example consider the following program fragment:

```
read (m);
read (n);
m = n + n;
```

In this example the program variable m is given a value by the read statement (a definition) and then the assignment (another definition), but is not referenced in between.

The *d-d* anomaly can indicate either: (1) that we have mistakenly used m to store the results of the computation n+n, or (2) we have read the wrong variable m as input. The data-flow analysis does not tell us what the fault actually is; it just tells us that there is potentially something wrong.

More generally what types of faults can data-flow analysis detect? Typically we can detect common types of programming mistakes, such as:

- typing errors
- uninitialised variables
- misspelling of names
- misplacing of statements
- incorrect parameters
- incorrect pointer references

4.2.2 Dynamic Data-Flow Analysis with Testing

We will update our terminology to be consistent with that of Rapps and Wuyeker. The first change to note is that Rapps and Wuyeker distinguish between two different uses of a variable:

Definition 31

(i) A **C-use** of a variable is a *computation use* of a variable, for example, y $=$ x $*$ 2;

(ii) A **P-use** of a variable is a *predicate use* , for example, if (x $<$ 2) ....

Secondly, we adopt the notation employed by Rapps and Wuyeker, as outlined in Definition 32 below.

Definition 32

(i) Let $d_n(x)$ denote a variable x that is assigned or initialised to a value at node (statement) n (**Definition**).

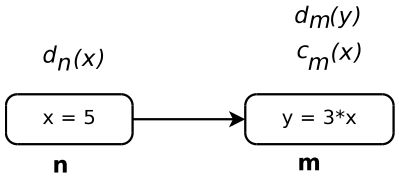
(ii) Let $u_n(x)$ denote a variable x that is used, or referenced, at node (statement) n (**Use**).

(iii) Let $c_n(x)$ denote a computational usage of the variable x at the node n (**Computational Use**).

(iv) Let $p_n(x)$ denote a predicate usage of the variable x at the node n (**Predicate Use**).

(v) Let $k_n(x)$ denote a variable x that is killed, or undefined, at a node (statement) n (**Kill**).

The data-flow annotations of Definition 32 are attached to nodes in the control-flow graph. The node **n** in the figure below



is annotated with a definition for x. This means that the variable x at node n is defined from the node n onwards along any path containing **n**. The node **m** is annotated with a *computational* use of x, and a definition of y.

All of the defines, uses, and undefines of a single variable are collated, and are represented using a *data-flow graph*. A data-flow graph is simply control-flow graph, except that its nodes are annotated with information regarding the definition, use, and undefinition of all of the variables used in that node.

Exercise 2 *Annotate the graph in Figure 4.8 with its definition, use, and undefinition information.*

We define the following for data-flow graphs.

Definition 33

(i) A *definition clear path* p with respect to a variable x is a sub-path of the control-flow graph where x is defined in the first node of the path p, and is not defined or killed in any of the remaining nodes in p.

(ii) A *loop-free path segment* is a sub-path p of the control-flow graph in which each node is visited at most once.

(iii) A definition $d_m(x)$ reaches a use $u_n(x)$ if and only if there is a sub-path p that is definition clear (with respect to x, and for which m is the head element, and n is the final element.

The aim is to define criteria with which to select and assess test suites. We will only look at some of the more popular of the full set of criteria discussed in the literature, and give some feel for their relative effectiveness.

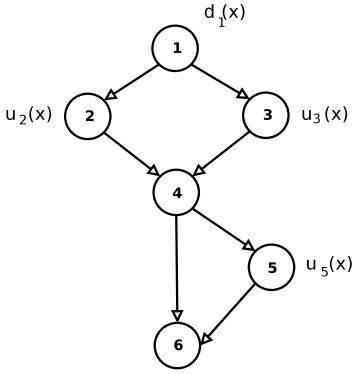
In particular we will be looking at the *data-flow path selection* criteria due to Rapps and Weyuker. The work is reported in [2].

The aim in data-flow based testing methods is to select test cases that traverse paths from nodes that *define* variables to nodes that *use* those variables, and ultimately to nodes that *undefine* those variables.

Coverage-Based Criteria

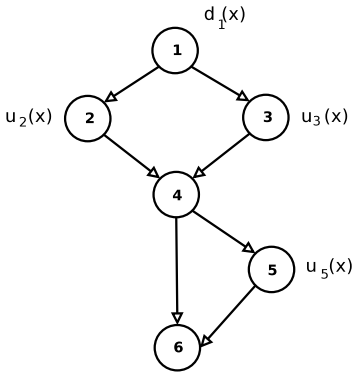
As with control-flow testing, the selection of test cases for data-flow testing involves achieving certain types of coverage on the graph, as defined by criteria. The most common forms of data-flow graph coverage criteria as the following.

1. **All-Defs** — For the All-Defs criterion we require that there is some definition-clear sub-path from *all definitions* of a variable to a single use of that variable. For example, consider the following data-flow graph:



   For a test suite to satisfy that All-Defs criteria, we would need to test at least one path from the single definition of x, to at least one use. A single test case is sufficient for this. The paths 1, 2, 4, 6 or the path 1, 3, 4, 5 would be satisfactory.

2. **All-Uses** — The All-Uses criteria requires some definition-clear sub-path from *all definitions* of a variable to *all uses* reached by that definition. For example, consider the following data-flow graph:



   The All-Uses criteria requires that we test $d_1(x)$ to each use and its successor nodes.

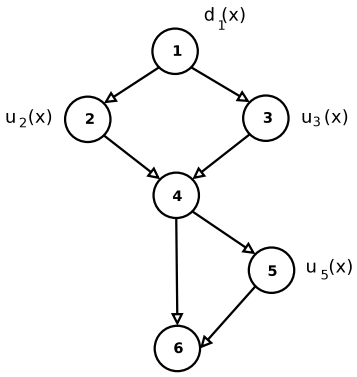
   (i) $d_1(x)$ to $u_2(x)$;

   (ii) $d_1(x)$ to $u_3(x)$; and

   (iii) $d_1(x)$ to $u_5(x)$.

   A test suite that traverses the paths 1, 2, 4, 5, 6 and 1, 3, 4, 6 are satisfactory under this criteria.

3. **All-Du-Paths** — Here **DU** stands for *definition-use*. The All-DU-Paths criterion requires that a test set traverse *all definition-clear sub-paths* that are cycle-free or simple-cycles from *all definitions* to *all uses* reached by that definition, and every successor node of that use.



The All-DU-Paths criteria requires that we test $d_1(x)$ to each use.

(i) $d_1(x)$ to a $u_2(x)$;
(ii) $d_1(x)$ to a $u_3(x)$;
(iii) both paths from $d_1(x)$ to $u_5(x)$.

Under this criteria, satisfactory paths are given by 1, 2, 4, 5, 6 and 1, 3, 4, 5, 6.

Recall from Definition 31, that a **P-use** of a variable is its use in a predicate, and a **C-use** of a variable is a use in a computation. Using these definitions, we can define the following additional data-flow test input selection criteria.

3. **All-C-Uses, Some-P-Uses** — The All-C-Uses, Some-P-Uses criteria requires a test set to traverse some definition-clear sub-path from each definition to each C-Use reached by that definition.

   If no C-Uses are reached by a definition, then some definition-clear sub-path from that definition to at least one P-Use reached by that definition.

4. **All-P-Uses, Some-C-Uses** — The All-P-Uses, Some-C-Uses requires that a test set to traverse some definition-clear sub-path from each definition to each P-Use reached by that definition and each successor node of the use.

   If no P-Uses are reached by a definition, then some definition-clear sub-path from that definition to at least one C-Use reached by that definition.

5. **All-P-Uses** — Some definition-clear sub-path from each definition to each P-Use reached by that definition and each successor node of the use

Tool Support

As with control-flow testing, manually applying data-flow testing is unlikely to yield outstanding results. However, there are parts of the process that are automatable. For example, the derivation of the data-flow graph is quite easily automated, due to its mechanical nature.

Derivation of test inputs from a data-flow graph would be considerably more difficult, especially for programs with non-linear domains, and for non-primitive data types. However, as with control-flow coverage, it is significantly easier to measure that a test suite has achieved some criterion by executing the suite over the program.

Coverlipse[1] is an open-source application that automatically determines whether a test suite achieves all-uses coverage, and provides feedback as to the paths that are missed by the test suite.

[1] See `http://coverlipse.sourceforge.net/`.

4.3 Mutation Analysis

> "*Programmers have one great advantage that is almost never exploited: they create programs that are* **close** *to being correct!*" — Richard DeMillo *et al.*[2]

Recall the squeeze function from Section 1.2, shown again below.

```
void squeeze(char s[], int c)
{
    int i,j;
    for (i = j = 0; s[i] != '\0'; i++) {
        if (s[i] != c) {
            s[j++] = s[i];
        }
    }
    s[j] = '\0';
}
```

Now consider and incorrect implementation of this function, in which the line

```
s[j++] = s[i];
```

is replaced with

```
s[j] = s[i];
```

That is, the variable j is never incremented, so the function places all references to c at position 0 in the array, and then terminates the string at position 0 (the last statement in the program). Clearly, this implementation contains a fault.

If we have a test suite for the squeeze function, and it is executed on the faulty version, one of two things can happen: 1) an error is encountered, and the programmer can debug the program to find the fault and repair it; or 2) an error is not encountered, and all test cases pass.

The latter occurs every day as part of every programmer's testing – whether they like to admit it or not is another story. Faults in programs go undetected because the test cases do not produce failures. When a fault is subsequently found, the test suite is (hopefully!) updated to include a test case that finds this fault, the fault is repaired, and the test suite is run again. The fact that the fault went undetected in the original test suite, and that the test suite is updated, means that the initial test suite was *inadequate*: there was a fault in the program that went undetected by it.

So far in these notes, we have discussed methods that aim to find faults by achieving coverage, but which give us little idea as to how good the resulting test suite is. In this section, we present *mutation analysis*: a method for measuring the effectiveness of test suites, with the side effect that we produce new test cases to be added to that test suite.

> **Definition 34** Given a program, a *mutant* of that program is a copy of the program, but with *one* slight *syntactic* change. The term "mutant" is an analogy for a biological mutant, in which small parts of a nucleotide sequence are modified by access to ionising radiation etc.

The example of the fault in the squeeze function is a mutant. In this case, the slight syntactic change is the removal of the ++ operator in the array index.

> **Definition 35**

[2] In R. DeMillo, R. Lipton, and F. Sayward, Hints on Test Data Selection: Help for the Practicing Programmer, *IEEE Computer*, 11(4):34–41, 1978.

(i) Given a program, a test input for that program, and a mutant, we say that the test case *kills* the mutant if and only if the output of the program and the mutant differ for the test inputs.

(ii) A mutant that has been killed is said to be *dead*.

(iii) A mutant that has not been killed is said to be *alive*.

The Coupling Effect

The theory behind the mutation analysis is related to the quote from DeMillo *et al.* at the start of this section. Programmers do not create programs at random, but rather write programs that are close to being correct, and continue to improve them so that they are closer to being correct over time. During this process, they learn the types of faults that programmers commonly make, and this experience is valuable in software testing. Most of these faults are either incorrect control flow of a program, or an incorrect computation, but the programs are close to their expected behaviour.

> **Definition 36** The *coupling effect* states that a test case that distinguishes a small fault in a program by identifying unexpected behaviour is so sensitive that it will distinguish more complex faults; that is, complex faults in programs are *coupled* with simple faults.

This is perhaps nothing Earth-shattering to anyone with experience in programming. If one is to look back over their repository logs, they would likely agree that many of the failures produced by their test suites are a result of one minor fault, or are so incorrect that almost any simple test case would uncover it. However, the coupling effect does have an impact on how we select test cases, whether via mutation analysis, equivalence partitioning, or any method. It implies that we should select test cases to find simple faults, and this will result in the same test cases finding larger ones.

The coupling effect has not been proven, and in fact, can not be proven. However, empirical studies of the the types of faults that are made in programs gives significant support to the theory.

The Coupling Effect, Mutants, and Testing

Recall the mutant of the squeeze program from the start of this section. We labelled any test suite that did not uncover this mutant as *inadequate*. But what relationship does this have to the test suites of programs that we are testing? In the squeeze example, we know where the fault is, so by not finding it, it is clear that the test suite is inadequate.

However, consider the following scenario. We are testing the original squeeze function with a test suite, and all of our test cases pass. To evaluate the quality of the test suite, we *deliberately* insert the fault into the program by removing the ++ operator in the array index, creating a *mutant* of the program, and then run the test suite again. If this produces a failure, then the fault has been uncovered, and the mutant is killed. However, if a failure is *not* produced, then we know that the test suite is inadequate, because there is at least one possible fault that it fails to uncover. If it fails to uncover this slight fault, then it would likely fail to uncover other faults. This is the process of *mutation analysis*.

In the latter case, the tester would then aim to find a test case that does kill the mutant, and add this to the test suite. Therefore, mutation analysis can be used to guide test input generation — *we must find a test case that kills every mutant* — as well as way of assessing test suite quality — *a test suite that kills more mutants than another is of higher quality*. Specifically, if test suite S kills all of the mutants that T kills, *plus some additional mutants*, then S subsumes T.

Systematic Mutation Analysis via Mutant Operators

Like other approaches to testing, mutation analysis is only effective if applied systematically. This is done using *mutant operators*.

Definition 37 A *mutant operator* is a transformation rule that, given a program, generates a mutant for that program.

Example 38 *Relational Operator Replacement rule for Java*

The *relational operator replacement* rule takes an occurrence of a relational operator, $<, \leq, >, \geq, =$, or $\neq$, replaces that occurrence with one of every other type of relational operator, and replaces the entire proposition in which that operator occurs with true and false.

Therefore, if the statement $\mathsf{if}(\mathsf{x} < \mathsf{y})$ occurs in a program, the following seven mutants will be created:

$\mathsf{if}(\mathsf{x} \leq \mathsf{y})$
$\mathsf{if}(\mathsf{x} > \mathsf{y})$
$\mathsf{if}(\mathsf{x} \geq \mathsf{y})$
$\mathsf{if}(\mathsf{x} = \mathsf{y})$
$\mathsf{if}(\mathsf{x} \neq \mathsf{y})$
$\mathsf{if}(\mathsf{true})$
$\mathsf{if}(\mathsf{false})$

This operator mimics some of the problems that programmers commonly make regarding the evaluation of Boolean expressions, which typically lead to incorrect paths.

Example 39 *Arithmetic Value Insertion rule for Java*

The *Arithmetic Value Insertion* rules takes an arithmetic expression and replaces it with its application to the absolute value function, the negation of the absolute value function, and *fail-on-zero* function, in which the fail-on-zero throws an exception if the expression evaluates to zero.

Therefore, if the statement $\mathsf{x} = 5$ occurs in a program, the following three mutants will be created:

$\mathsf{x} = \mathsf{abs}(5)$
$\mathsf{x} = -\mathsf{abs}(5)$
$\mathsf{x} = \mathsf{failOnZero}(5)$

This operator is designed to enforce the addition of test cases that consider the case in which every arithmetic expression to evaluate to zero, a negative value, and a positive value. This mimics some of the problems that programmers commonly make, such as dividing by zero.

Mutants are *syntactic* changes to a program, so mutant operators are dependent on the syntax of programming languages. Therefore, the mutant operators of one programming language may not necessarily apply to other languages.

Remark 40 It is important to note that mutant operators must produce mutants that are *syntactically valid*. It is possible to create mutants of programs that are not syntactically valid, however, in most programming languages, a compiler will detect these, so they cannot be killed by a test case. One could consider that they are killed by the compiler, however, syntactically invalid mutants will always be caught by a correct implementation of a compiler, so they give us no insight into test suite quality or test input generation. Even if a language is dynamically interpreted, a good collection of mutant operators will ensure that any statically invalid programs are killed.

In addition to the two mutant operators outlined above, Ammann and Offutt [1] define the following mutant operators for the Java programming language.

- *Arithmetic Operator Replacement*: Replace each occurrence of an arithmetic operator +, −, *, /, **, and % with each of the other operators, and also replace this with the left operand and right operand (for example, replace x + y with x and with y).
- *Conditional Operator Replacement*: Replace each occurrence of a logical operator &&, ||, &, |, and ^ with each of the other operators, and also replace this entire expression with true, false, the left operand, and the right operand.

- *Shift Operator Replacement*: Replace each occurrence of the shift operators $<<$, $>>$, and $>>>$ with each of the other operators, and also replace the entire expression with the left operand.
- *Logical Operator Replacement*: Replace each occurrence of a bitwise logical operator $\&$, $|$, and $\hat{}$ with each of the other operators, and also replace the entire expression with the left and right operands.
- *Assignment Operator Replacement*: Replace each occurrence of the assignment operators +=, $-$=, *=, /=, %=, &=, |=, ^=, $<<$=, $>>$=, and $>>>$= with each of the other operators.
- *Unary Operator Insertion*: Insert each unary operator, +, $-$, !, and $\sim$ before each expression of the correct type. For example, replace $\mathsf{if}(\mathsf{x} = 5)$ with $\mathsf{if}(!(\mathsf{x} = 5))$.
- *Unary Operator Deletion*: Delete each occurrence of a unary operator.
- *Scalar Variable Replacement*: Replace each reference to a variable in a program by every other variable of the same type that is in the same scope.
- *Bomb Statement Replacement*: Replace every statement with a call to a special Bomb() function, which throws an exception. This is to enforce statement coverage, and in fact, only one call to Bomb() is required in every program block.

To systematically generate mutants, each of these operators is applied to every statement of a program to which it is applicable, generating a number of mutants for the program. Each of these mutants is then tested using the test suite, and the percentage of mutants killed by the test suite is noted.

> **Definition 41** The *mutation score* of a test suite for a program is the percentage of the mutants killed by that test suite, calculated by dividing the number of killed mutants by the number of total mutants.

Equivalent Mutants

A major problem with mutation analysis is the *equivalent mutant* problem.

> **Definition 42** Given a program and a mutation of that program, the mutant is said to be an *equivalent mutant* if, for every input, the program and the mutant produce the same output.

An equivalent mutant cannot be killed by any test case, because it is equivalent with the original program. As an example of an equivalent mutant, consider the squeeze function. Using the *Arithmetic Value Insertion* rule, the line

```
s[j++] = s[i];
```

can be mutated to

```
s[j++] = s[abs(i)];
```

in which abs is the absolute value function. However, the variable i only ever takes on values between 0 and the size of the array, so abs(i) will be equivalent to i irrelevant of the test inputs that we choose. Therefore, this mutant is equivalent to the original program, and cannot be killed.

Equivalent mutants pose problems because they are impossible to kill, and difficult to detect. That is, once a test suite has been run over a collection of mutants, it is difficult to determine which mutants have not been killed due to the test suite being inadequate, or due to them being equivalent. Computing whether two programs are equivalent is undecidable, so a manual analysis needs to be undertaken for many instances, although some automated techniques exist that apply heuristics to eliminate equivalent mutants.

The equivalent mutant problem implies that, for many programs, an *adequate* test suite — that is, one that kills every mutant — is unachievable.

In many practical applications of mutation analysis, a *threshold* score is targeted — for example, the tester aims to kill 95% percent of the mutants, and continues trying to kill mutants until that threshold is reached.

Exercise 3 *Using the mutant operators from above, define all of the mutants for the* squeeze *function. Identify which of these mutants are equivalent.*

Tool Support

Mutation analysis is an expensive process. Firstly, it can can generate quite a large number of mutants. In fact, the number of mutants grow exponentially with the size of the program. Generating these mutants manually is not feasible for anything other than small programs. Secondly, it requires the tester to execute the test suite over every mutant, and most likely to iterate this execution a number of times.

For these reasons, a lot of work has been put into tools for automatic mutant generation, such as MuJava[3], a mutant generator for Java that uses the above rules, Jumble[4], a Java mutant generator that mutates bytecode instead of source code, and PITest [5], which also works on Java bytecode. These tools automatically generate the mutants for a program given its sourcecode or bytecode, and, given a test suite, automatically execute the mutants, and produce a report outlining the mutation score, and which mutants are still alive.

Unfortunately, the equivalent mutant problem is not solved by the above tools. However, research into mutation analysis is leading to cheaper ways of automatically eliminating equivalent mutants using constraint solvers and compiler optimisation techniques. Applying these tools in practice is a long way off yet.

At this cost comes benefit: mutation testing is considered to be the most successful test coverage criterion for finding faults. In the next section, we compare the three coverage criteria presented in this chapter.

4.4 Comparing Coverage Criteria

A question that often arises in practice is: *which criteria gives the best coverage*?

The way to compare these criteria is to defined a specific relation between the criteria. The relation in question is called *subsumption* and is defined as follows.

Definition 43 Criterion A subsumes criterion B iff for any control-flow graph P:

$$P \text{ satisfies } A \implies P \text{ satisfies } B.$$

Criteria A is equivalent to criteria B iff A subsumes B, and B subsumes A.

The *subsumes* relation is transitive, therefore, if A subsumes B, and B subsumes C, then A subsumes C.

How can we compare these criteria?

Both data-flow criteria and coverage criteria select a set of paths that must be traversed by the test cases. To compare across different criteria Frankl and Weyuker have compared the paths that each criteria selects. Note, however, that the set of paths that satisfy a criterion are not necessarily unique. The results are shown in Figure 4.10.

It is straightforward to see the relationships between the different data-flow techniques, and between the different control-flow techniques. For example, branch coverage (**All-Edges**) subsumes statement coverage (**All-Nodes**) because every statement is located within a branch.

The relationships between the different control- and data-flow criteria are less clear to see. Condition coverage does not subsume statement coverage. This can be illustrated by the example if (a && b). Condition coverage mandates that a test suite must exercise a as both true and false, and b as both true and false. This can be achieved with two test cases: (a == true, b == false), and (a == false, b == true). However, these test cases never execute the case that a and b are *both* true, therefore, the statements in that branch are not executed.

[3] See `http://cs.gmu.edu/~offutt/mujava/`.
[4] See `http://jumble.sourceforge.net/`.
[5] See `http://pitest.org/`

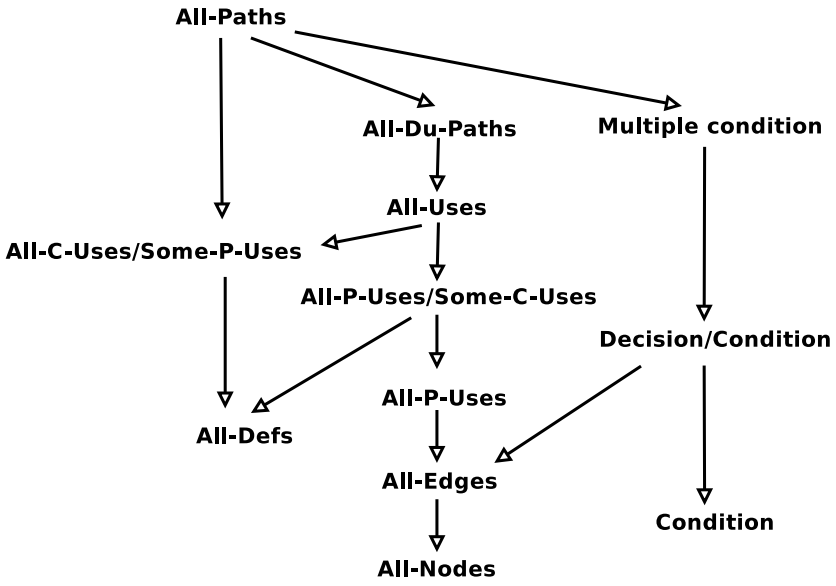


Figure 4.10: Subsumption relations between sets of test cases chosen by various test case selection criteria.

Multiple condition coverage and the criteria that it subsumes do not subsume any of the data-flow criteria. This can be illustrated with the following simple program:

```
if (y == 1) {
    x := 2;
}

if (z == 1) {
    a := f(x)
}
```

in which f is some function. In this example, multiple condition coverage enforces that $y == 1$ and $z == 1$ are both executed for the true and false cases. To achieve any of the data-flow criterion, at least one test must execute a definition-clear path from the statement $x := 2$ to $a := f(x)$. However, multiple condition coverage does not enforce this, therefore, it does not subsume any of the data-flow criteria.

Similarly, none of the data-flow coverage criteria subsume any of the coverage metrics related to conditions or decisions. If we consider the weakest of these, condition coverage, the none of the data-flow criteria will evaluate the following statement to false:

```
if (x == 1) {
    a := f(x)
}
```

because there is no block related to the false case, and therefore no variable uses in it.

Decision/condition coverage clearly subsumes both branch and condition coverage, as it is a combination of the two. The *subsumes* relation is transitive, therefore, decision/condition coverage also subsumes statement coverage, because branch coverage does. Multiple-condition coverage subsumes decision/condition coverage, and path coverage (if possible) subsumes branch coverage.

Mutation analysis has not been discussed so far. It is difficult to evaluate because mutant operators are defined specific to programming languages. However, one can easily take the operators defined by Ammann and Offutt for Java, described earlier, and apply them to many programming languages.

These operators subsume many of the criteria in Figure 4.10, including multiple-condition coverage, and All-Defs

coverage. This can be proved in these cases. For example, any test suite that kills all non-equivalent mutants generated by the *Bomb Statement Replacement* rule is guaranteed to achieve statement coverage (or node coverage in a control-flow graph), because every statement is replaced by a call to `Bomb()`. Similarly, any test suite that kills all non-equivalent mutants generated using the *Conditional Operator Replacement* rule is guaranteed to achieve multiple-condition coverage.

In Section 5.2.2. of [1], Ammann and Offutt prove that mutation analysis is stronger than many of the coverage criteria in Figure 4.10. They hypothesise that even though some of them are not subsumed by mutation analysis, it is likely that specific mutant operators could be derived to achieve this, but that there is little benefit in doing so.

4.4.1 Effectiveness of Coverage Criterion

The effectiveness of these criteria is important to consider. If we go to the effort of measuring coverage and adding tests to ensure that we achieve coverage, we'd like to know that we are adding value to our test suite.

Several studies have looked into the effectiveness of test coverage criteria. For example, a 2014 study 3 looked into the correlation between coverage and fault-finding ability, considering statement coverage, branch coverage, and modified-condition coverage using 31,000 test suites, and found that there is a low correlation between test suite coverage and fault-finding effectiveness. Further, they found that stronger forms of coverage provide very little value. Another 2014 study 4 showed that mutation score provides is much better correlated with fault-finding ability, which is unsurprising because the mutation score is high if a test suite finds seeded faults.

These studies confirmed a long-held view in software engineering that structural coverage criteria, such as control- and data-flow criteria, should be used to identify areas of the program that remain untested, and not as measure of the quality of the test suite.

In short, structural test coverage is a *necessary but not sufficient* method for software testing.

4.5 References

[1] P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008.

[2] P. G. Frankl and E. J. Weyuker, An Applicable Family of Data-Flow Testing Criteria, *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[3] L. Inozemtseva and R. Holmes, Coverage is not strongly correlated with test suite effectiveness, *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014.

[4] R. Just, et al., Are mutants a valid substitute for real faults in software testing?, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014.

Chapter 5

Test Oracles

In this chapter, we will discuss *test oracles*. Recall from Section 1.5 that a test case consist of three elements:

1. A test input or sequence of test inputs;
2. An expected output or sequence of expected outputs; and
3. The testing environment.

The normal procedure for executing a test case is to execute the program using the inputs in the test case, record the results, and then to determine if the outputs obtained are failures[1] or not.

Who or what determines if the results produced by a program are failures? One way is for a human tester to look at the result of executing the test input and the expected results and decide if the program has failed the test case. In this case the human tester is playing the role of a *test oracle*.

A test oracle is someone or something that determines whether or not the program has passed or failed the test case. Of course, it can be another program that returns a "yes" if the actual results are not failures and "no" if they are.

Definition 44 A *test oracle* is:

- a program;
- a process;
- a body of data;

that determines if actual output from a program has failed or not.

Ideally an oracle should be automated[2] because then we can execute a larger volume of test cases and gain greater coverage of the program, **but** this is often extremely hard in practice.

5.1 Active and Passive Test Oracles

An automated oracle can be placed into one of two categories:

Active oracle A program that, given an input for a program-under-test, can generate the expected output of that input.

[1] This is a *failure* in the sense of Section 1.3.2

[2] Automated means that we can execute the oracle with no human intervention.

Passive oracle A program that, given an input for a program-under-test, and the actual output produced by that program-under-test, verifies whether the actual output is correct.

Passive oracles are generally preferred. This is for two main reasons.

Firstly, passive oracles are typically easier to implement than active oracles. For example, consider testing a program that sorts a list of numbers. It is considerably easier to check that an output produced by the program-under-testis a sorted list, than it is to sort this list. This not only saves the tester some time, but also means that there is less chance of introducing a fault into the oracle itself. If the active oracle is required to simulate the entire program-under-test, it may be as difficult to implement, and therefore, is just as likely to contain faults of its own.

The second reason that passive oracles are preferred is because they can handle *non-determinism*. Recall from Section 1.2, that a program is non-deterministic if it can return more than one output for a single input. If an active oracle is used, there is a good chance that the output produced by the program-under-test will be different to the output produced by the active oracle. However, if we use a passive oracle, which simply checks whether the output is correct, then non-determinism is not an issue.

5.2 Types of Test Oracle

In general, to design a test oracle, there are several types of oracle that can be produce, which are categorised by the way their are derived, or the way they run.

5.2.1 Formal, executable specifications

Formal specifications written in a tight mathematical notation are better for selecting and creating testing oracles than informal specification written in a natural language. They can be used as active oracles by generating expected output using a simulation tool, or as passive oracle, by proving that the specification is satisfied by the input and the actual output. These are unlikely in practice, and are mostly reserved by high integrity applications such as safety-, security-, or mission-critical systems.

5.2.2 Solved examples

Solved examples are developed by hand, or the results from a test input can be obtained from texts and other reference works. This is especially useful for complex domains, in which deriving the expected output automatically requires a process as complicated as the program itself, and deriving it manually requires expertise in a specific area that a test engineer is unlikely to have.

Data in the form of tables, documents, graphs or other recorded results are a good form of testing oracle. The test input and actual results can be looked up in a table or data-base to see if they are correct or not.

These types of oracles have the disadvantage that the inputs chosen are restricted to the examples that we have access to. Despite this, they are common due to their abundance in many fields.

5.2.3 Metamorphic oracles

In some cases, we can use certain metamorphic properties between tests to check each other. For example, if we want to test a function that sorts a list of numbers, then we can make use of the fact that, given a list, any permutation of that list will result in the same output of the sort function (assuming a non-stable sort).

To do metamorphic testing, we generate an input for a program, execute this input, and then generate another input whose output will be related to the first. In the sorting example, we can generate a test input as a list of numbers, and then randomly permute the elements of the list to get a new list. Then, we execute *both* tests on the sort function, and compare their output. If their output is different, then we have produced a failure.

Another example is a program for finding the shortest path between two nodes on a graph. We can select any two nodes on the graph and run the program, returning a path. To check if this path is correct, we can select any two nodes on that path and run the shortest path program on those two nodes. The resulting path should be a sub-path of the first path.

The existence of metamorphic properties for programs is surprisingly common, and metamorphic oracles have been used to test many numerical programs, but also many applications in non-numerical domains as well, including bioinformatics, search engines, machine learning, medical imaging, and web services.

5.2.4 Alternate implementations

An alternate implementation of a program, which can be executed to get the expected output. This is not ideal because experience has shown that faults in different implementations of the same specification tend to be located on the same inputs. Therefore, an alternate implementation is likely to have the same faults as the program-under-test, and some faults would not be detected via testing as a result.

One approach that has shown to be useful is to provide a *partial*, *simplified* version of the implementation, which does not implement the full behaviour of the program-under-test, and does not consider efficiency or memory.

For example, if we are an efficient sorting algorithm, then we can restrict the oracle (the alternative implementation) to only sorting lists of integers whose values, when sorted, form a complete integer sequence; e.g., $10, 11, 12, 13, 14, 15$. To perform the sort, the oracle needs to only find the lowest and highest element in the list using a linear search, and then return a list with the lowest element at the first index, the highest element at the last, and the corresponding elements in between. This is partial, and perhaps not efficient, but it results in a list that is a sorted version of the inputs, and is less likely to contain a fault than the original sorting algorithm due to its reduced complexity.

Such an approach restricts the test inputs that can be used, but is often sufficient to find many faults in a system.

5.2.5 Heuristic oracles

Perhaps the most widely-used types of automated oracles being used today are *heuristic oracles*. These are oracles that provide *approximate* results for inputs, and tests that "fail" must be scrutinised closely to check whether they are a true or false positive.

The trick with heuristic oracles is to find patterns in complex programs — that is, patterns between the inputs and outputs — and exploit them. For example, a very simple heuristic of databases is that, when a new record is inserted into a table, the number of records in that table should increase by 1. We can run thousands of tests inserting single records, and checking that the number of rows in table increases by 1. This is not complete though, because we are not checking that the contents of the row are accurate.

As a more realistic example, consider an oracle for checking a system that calculates driving directions for a GPS-enabled device. If the algorithm finds the shortest path between the start point and the destination, a complete oracle would need to check that this is indeed the shortest path. However, to check this fully, our oracle would have to re-calculate the shortest path as well, which would likely be as complicated as the original algorithm, and therefore just as prone to faults. Instead, we can use a heuristic that states that the shortest path should be within, e.g. 1.5 times the distance of a straight line between the two points. Anything outside of this could signal a fault. This is clearly not complete: the distance between two points may be small, while the shortest path via a road network may have to take a bridge that is far away from the destination.

In fact though, all oracles are heuristic, in that none of them really replicates the expected behaviour of the corresponding program. However, we use the term *heuristic oracle* to refer to oracles that are designed based on some heuristics about the software under test.

5.2.6 The Golden Program!

The ultimate source for a testing oracle but rare in practice. The golden program is an executable specification, a previous versions of the same program, or test in parallel with a trusted system that delivers the same functionality.

Still, the golden program is not a pipedream. In industry, it is not uncommon to use the previous release of a piece of software as a test oracle.

5.3 Oracle derivation

In these notes, we will not consider how to derive oracles. Current industry practice leaves much of the test case generation, including the oracle, up to human testers, who typically derive the oracles by looking at the specification and design of the artifact that are testing. In many unfortunate cases, the tester is left to guess what the behaviour of the software should be.

State-of-the-art testing includes *model-based testing*, which is used to automate both input and oracle generation. Using model-based testing, rather than the test engineer deriving test cases, he/she derives a *model* of the expected behaviour of the program-under-test, using some formal, executable language. From this model, test inputs are generated automatically (using the types of criteria that we discuss in these notes), and the expected outputs for those inputs are calculated by simulating the model. This can be seen as a cross between the first and last types of oracle. The model can be seen as an abstract alternative of the implementation, which is both formal and executable. However, unlike other alternate implementations, the higher level of abstract means that the likelihood of faults being located on the same inputs is reduced. Empirical evidence demonstrates that model-based testing is, in general, no more expensive than manual test case generation, and in many cases, is significantly more efficient, and is as successful for locating faults.

Chapter 6

Testing Modules

In the previous chapters, we have discussed how to select test inputs to maximise the chances of uncovering faults in a program. Examples in these chapters have been single functions, with only parameters as input, and return values as output. In this chapter, we discuss the testing of program with state; that is, testing *modules*. Modules present some different problems with regards to testing when compared to single functions.

6.1 State and Programs

During the years proceeding the wide-spread adoption of third-generation programming languages, computer programs started becoming more and more complex. Practitioners realised that, instead of allowing any part of a program to manipulate any part of data, if access to data was provided via a clear and unambiguous interface made up of functions on that data, then complex program could be understood more readily.

Typically, the data, called the *state*, is manipulated and accessed via a collection of *operations* on that state. Collectively, these are referred to as a *module*[1]. Using modules enforces a separation of interface and implementation, allowing programmers to think of these collections of operations as a black box. Specific details about the data is hidden from the user of the module, and a change in the underlying data structure has no impact on the program that uses it as long as the interface remains the same.

Operations are meaningless when separated from their state and treated in isolation. For example, an operation may require the module to be in a specific state before it can be executed, and that state can only be set by another operation (or combination of operations) in the module.

Object-oriented programs are special classes of state-based programs, in that multiple instances of the data can be created.

6.2 Testability of State-Based Programs

Recall from Section 1.3.5 that the *testability* of a program is defined by the *observability* of the program, and the *controllability* of the program. State-based programs present some issues with regards to these two measures, even for APIs.

For example, consider a module that allows the manipulation of and access to a *stack*. The operations of the class allow us to push an element on to the stack, view the top of the stack, pop the top element from the stack, and check whether the stack is empty. The specification of the class is allows:

[1] Strictly, a module does not require a state, but can be simply a collection of related operations.

Initially, the stack is empty — it contains no elements. When created, the calling program determines a maximum size for the stack, passed as a parameter to the constructor.

An element can be *pushed* onto the stack if the size of the stack is not already equal to the maximum size, as determined when initialised. If full, pushing another element will result in the exception StackFullException.

The top element of the stack can be *popped* from the stack provided that the stack is not empty. Pushing an element and then popping the stack will result in the stack before pushing. If the stack is empty, popping will result in the exception StackEmptyException.

The calling program can see the value of the top of the stack, but no other parts of the stack. If the stack is empty, peeking at the top of the stack will result in the exception StackEmptyException.

Finally, the stack can be checked to see if it is empty, and can be checked to see if it is full. These operations allowing the calling program to avoid exceptions.

Figure 6.1 shows an implementation of such a module in Java, in which the stack has a maximum size, which is determined by the calling program when creating the stack data type. The variables prefixed with an underscore (_) as those that are encapsulated by the class, and are not directly controllable or observable.

Using the techniques discussed in the previous chapters, we can derive some test cases to test the operation that pushes an element from the stack. Any good test suite will include at least equivalence classes for an empty stack, and a stack with at least one element. After selecting values from these equivalence classes, we can derive the executable test cases.

However, access to the stack data is hidden behind the interface. The push operation does not take a parameter for the stack, meaning that the push operation cannot be *controlled* via its interface. Furthermore, once an element is pushed onto the stack, we cannot directly *observe* the value of the stack. Therefore, the operations as part of the stack module are not easily controllable or observable.

This reduction in controllability and observability means that unit testing in isolation is not possible for the operations of the stack module. In cases such as this, the other operations that make up the module may be required to test an operation, so we have a case in which unit testing and module testing can be considered as the same activity.

One way around these testability issues is to *intrusively* test the module. That is, the tester modifies the program code to give access to the data type that is hidden, therefore breaking the information hiding aspect of the module. The program is tested, and the data type is hidden again. This has serious drawbacks:

- it does not test the implementation that is to be used, but an altered version of it;
- if the underlying data type in the module changes, then the tests must be changed as well; and
- it ignores the fact that operations acting on the same data as inherently linked, and must be tested together. One could argue that the inter-dependencies could be tested as well, but with modules such as the stack class, the high-level of dependency between operations implies that testing an operation in isolation would give us little benefit.

Rather than intrusively testing the module, we use a *non-intrusive* method: we use the operations defined by the module to set the state of the module to test value that we want, perform the necessary test, and then use the operations again to query the result of the test. In the example of the stack class, to test the behaviour of pushing an element onto a stack containing one element, we first push an element onto an empty stack using the push (setting up the state for our test), then push another element for our test, then use the top and pop operations to see the state of the module.

Note that many object-oriented languages provide us with constructs that are somewhat in between intrusive and non-intrusive testings. For example, in the Java programming language, variables inside classes can be declared as *protected*, which means that if we can inherit the class, then we can access the variables in the class. This allows us to passively observe the values of variables without changing the module itself. Other tools provide non-intrusive ways to observe data, such as inserting breakpoints in programs, and allowing the tester to observe the state of the program at that breakpoint.

```
public class Stack
{
    private object _stack[];
    private int    _maxsize = 0;
    private int    _top = 0;

    public  void Stack(int maxsize)
    {
        _maxsize  = maxsize;
        _top   = 0;
        _stack = new Object[maxsize];
    }

    public void push(Object n)
        throws StackFullException
    {
        if (_top == _maxsize)
          throw new StackFullException();
        else {
          _stack[_top] = n;
          _top++;
        }
    }

    public void pop()
        throws StackEmptyException
    {
      if (_top =< 0)
         throw new StackEmptyException();
      else
         _top--;
    }

    public Object top()
        throws StackEmptyException
    {
        return _stack[_top];
    }

    public boolean isEmpty()
    {
        return (_top == 0);
    }

    public boolean isFull()
    {
        return (_top == _maxsize);
    }

}
```

Figure 6.1: A Java stack class

6.3 Unit Testing with Finite State Automata

The way that many modules, especially those that implement abstract data types, are abstracted is to envisage them as *finite state automata*.

> **Definition 45** A *finite state automaton* (FSA) or *finite state machine* is a model of behaviour consisting of a finite set of states with actions that move the automata from one state to another.

With regards to a module, each state in a FSA corresponds to a *set* of states in the module it is modelling. The transitions between states correspond to the operations in the module. Transitions can be prefixed with predicates specifying the preconditions that must hold for them to take place.

FSAs are useful for modelling behaviour of programs, and can be used to derive test cases for such programs. The states of the automaton derived from subsets of the states of the data encapsulated in the module, and the transition of the automaton are the operations of the module. Test cases are selected to test sequences of transitions in this automaton.

Constructing a FSA

To construct a finite state machine, we perform the following steps.

Step 1 — identify the states of the FSA. Each state in the FSA corresponds to a set of states in the module. Intuitively, we want to consider states with same effect on operations as being *equivalent* and collect them together into a single FSA state. The situation is analogous to the situation in equivalence partitioning where every element of an equivalence class has the same chance of finding an error. Therefore, we use the techniques described in the previous chapters to derive the states that we want to test.

Step 2 — identify which operations are enabled in a state and which operations are not enabled in a state — an operation is enabled if it can be called safely without error in a given state.

For example, the pop and top operations are not enabled if the stack is empty, and the push operation is not enabled if the stack is full.

From this information, we identify the *start* state(s), the *exit* state(s) (those that have no enabled operations).

Step 3 — is to identify the source and target states of every operation in the model. To do this, we take every state and every operation that is enabled in that state, and calculate the state that results in applying that operation.

For example, if we derive a state specifying the stack as empty (state s_0), and a state specifying the stack having one element (state s_1), then the push operation links s_0 to s_1, the pop operation links s_1 to s_0, and the isEmpty operation links both states to themselves. If modelling exceptions, then applying the pop operation in the empty state will result in the state StackEmptyException.

Example 46 *Deriving the stack FSA*

As a first example, consider a simple Stack class that we wish to test. An informal specification for this is given in Section 6.2, and an implementation of that specification is given in Figure 6.2.

Firstly, we want to derive the states of the FSA. Intuitively, we want to consider states with same effect on operations as being *equivalent* and collect them together into a single FSA state. The equivalence partitioning techniques can be used to derive this. Although equivalence partitioning is an *input* partitioning technique, the state can be considered as input to any operation that uses it, but it is input that is not passed as a parameter.

Instead of looking at the input domain we look at the states of a module and ask what states can be considered equivalent if we want to find faults in the module operations. We do this for the Stack class in Table 6.1.

The *initial* state of this module is the undefined stack. Alternatively, one could remove the undefined state, and instead specify that the initial state is the empty state. The exit states are the exception states.

State	Description	Enabled Operations
State 1	An undefined stack	Stack constructor
State 2	A defined stack that is empty; that is, a stack with no values	push, isEmpty, isFull
State 3	A stack that is full; that is, it has the maximum number of elements	pop, top, isEmpty, isFull
State 4	A stack that is defined but is neither empty nor full	push, pop, top, isEmpty, isFull
State 5	A stack that is in the StackEmptyException state; that is, the caller has attempted to pop or see the top of an empty stack	none
State 6	A stack that is in the StackFullException state; that is, the caller has tried to push an element onto a full stack	none

Table 6.1: States for the Stack module

The exit states depend on the the specification for the module. For example, popping an empty stack throws an exception, however, an alternative specification of the stack may *assume* that the stack is not empty when elements are popped, therefore, the behaviour of popping an empty stack is undefined, and cannot be meaningfully tested. In this case, there are no exit states (there is always one operation that can be executed in any state).

Finally, we derive the transitions between the states. The final FSA for the stack module is shown in Figure 6.2. The transitions of isEmpty and isFull are omitted because they are enabled in all states (except the undefined state) and always loop back to the same state from which they started.

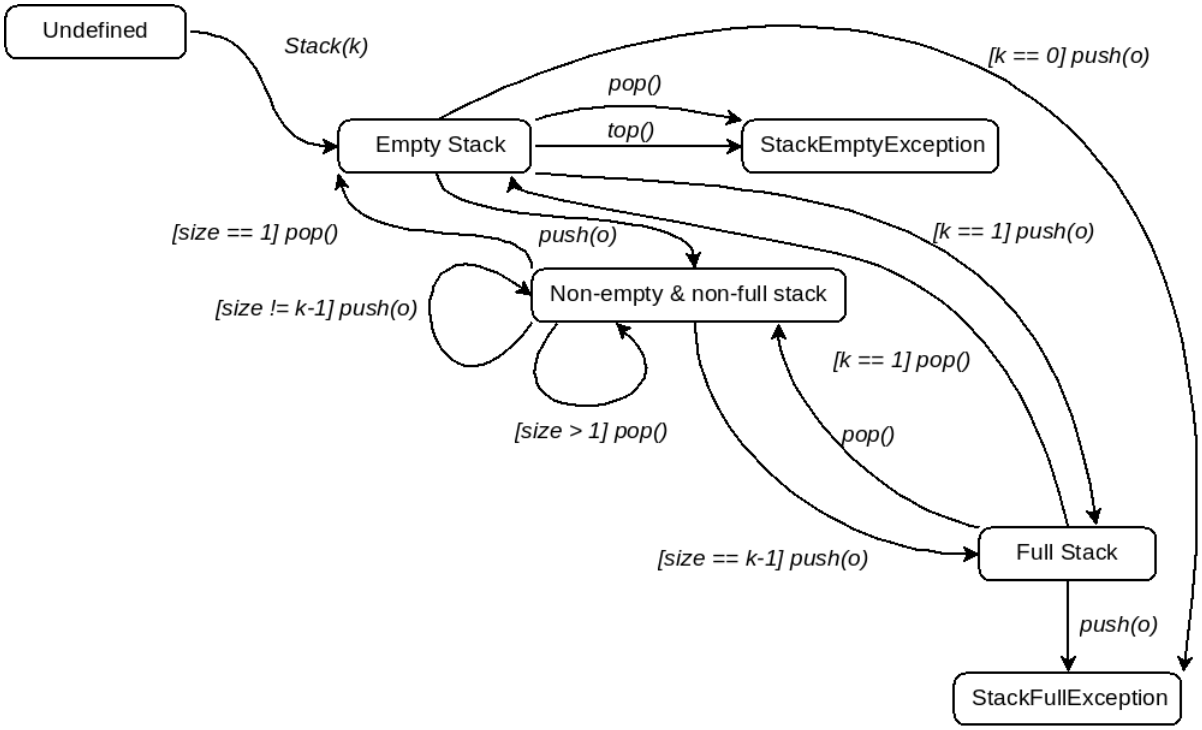


Figure 6.2: Finite State Automaton for the Stack ADT

In this figure, the predicate in the square brackets on the transition labels represent the conditions that must hold for that transition to take place. Note that these are not the preconditions of the operations themselves, but merely the conditions that must hold for that operation to change to the destination state.

Note the push transition directly from the empty state to the full state, with the condition k == 1, is for when the maximum size is 1, in which case there is no state in which the stack can be non-empty and non-full. Similarly for the reverse action: popping the stack from a full state, and for pushing onto a stack of maximum size 0, which leads to an exception state.

6.3.1 Deriving Test Cases from a FSA

Recall from Section 6.2 that the observability and controllability of modules is low due to their hidden information. We discussed that fact that to set up a state for a test input of an operation may require us to execute other operations in the module.

The FSA of a module gives us the information that is required to do so. If the states in the FSA represent the input states that we wish to test, then the transitions that move the state from the initial state to each state represent the operations that need to be executed.

> **Definition 47** A *path* from a state s_0 to a state s_n in the automaton is a sequence of transitions $t_1 t_2 \ldots t_n$ such that the source of t_1 is s_0 and the target of t_n is s_n.

The aim is to generate *test sequences* such that the paths in the automaton are exercised. A test case is no longer a single function call but may require a sequence of operation calls to force the module along a certain path.To derive these sequences, we define *traversal criteria* for a FSA. Broadly, there are three criteria that we can use:

(i) *State coverage* — each state in the FSA must be reached by the traversal.

(ii) *Transition coverage* – each transition in the FSA must be traversed. This subsumes state coverage.

(iii) *Path coverage* — each path in the FSA must be traversed. This subsumes state coverage, but is impossible to achieve for a FSA with cyclic paths.

State coverage is clearly inadequate, because there are transitions in the FSA that will not be traversed, and therefore, operations that will not be tested for the test states that were derived. Path coverage is impossible for FSAs with cycles, so this is infeasible in many cases. Transition coverage is feasible, straightforward to achieve, and it tests every test state that was derived, so this is generally the most practiced traversal technique.

Using transition coverage, we repeatedly derive test sequences starting at the initial node until all transitions have been covered at least once. In some cases, this can be done using one long sequence, but in others, this is not possible; for example, the stack FSA as two exit states, and one of these has two input transitions, which together imply that there must be at least three sequences. Often, it is more efficient to derive many short sequences than a few long sequences.

Intrusively Testing the State Transition Diagram

Now we an look at intrusive and non-intrusive methods for testing an automaton.

As discussed earlier, *intrusive testing*, in these notes at least, will mean adding testing code to the module to measure and monitor the internal states of the module as it is tested. The problem with this kind of testing is that it breaks encapsulation.

The points at which to insert testing code into a module are to observe the values of private state variables, observe the values of private operation calls and even observe intermediate states in module operations.

In an object-oriented programming language, this is not as obvious as it sounds because of the problem of inheritance. Consider the class Node for implementing a doubly linked list in the following example.

```
public class Node
{
    private Object _data;
```

```
    private Node   _previous, _next;
}
```

To observe the state of Node we need to get access to the values of the _data instance variable. However, because _data is an instance of the Object class, this means that we need to ensure that there are operations to access the private state elements of all the objects in the system that inherit from Object[2]; that is, we would need to extend all elements of the hierarchy with testing code. In this case there is a great deal of additional code to write which may well further impact the properties of the entire system.

In languages like C and C++ we can use the pre-processor to compile test code into the executable when required, or to omit the testing code when the program is to be released.

As a result of these problems, systematic intrusive testing is difficult to achieve, in fact, it is often impossible for some languages if we do not have access to the source. Many of the solutions are hacks that give the testers some benefit that outweighs the breaking of the encapsulation.

The best cases are languages that provide semi-private access, such as the protected keyword in Java, or the friend keyword in C++. These give testers the ability to read and write to the variables without changing the program-under-test.

Non-intrusive Testing the State Transition Diagram

There will be occasions when we simply do not have access to the internal structure of a module or its hierarchy. In this case, or in the case where you simply need to test a module without adding code to a module, then we have *non-intrusive* testing.

The idea is to test each of the transitions in the testing automaton implicitly by using other operations in the module to examine the results of a transition. Lets start by dividing up the operations in a module as follows:

Initialisers	The operations that initialise the module, such as the Stack constructor.
Transformers	Operations to change the state of the module, such as pop operation in the stack example.
Observers	Operations to observe the module state, such as the isEmpty and top operations in the stack example.

Table 6.2: One classification of the operations of a module.

Recall that in the FSA for the stack module, shown in Figure 6.1, the IsEmpty, isFull, and top operations were omitted because they do not move the automaton to a new state. This is because these operations are *observers* — with the exception of top moving to the StackEmptyException state.

The technique for testing involves deriving test sequences, using initialisers and transformers, to move the automaton to the required state. At each state, observe the value of the module using the observer operations. The only case where the observer operations are not used is in the exception states, because the execution of the module is considered to have terminated. Alternatively, and depending on the programming language, these exception states could be remove from the FSA, and considered as observer behaviour. Many programming languages support the catching of exceptions, thereby allowing the execution to continue.

Example 48 *Traversing the stack FSA*

For the stack automaton in Figure 6.2, we can obtain transition coverage using the following test sequences:

[2]In the case of the type Object in Java this can mean a large number of classes in theory, but in practice it is not often that large.

Sequence	**End State**
$\mathsf{Stack(0); push(o)}$	StackFullException
$\mathsf{Stack(1); push(o); pop(); pop()}$	StackEmptyException
$\mathsf{Stack(k); top()}$	StackEmptyException
$\mathsf{Stack(k); push(o)^k; pop()^k; push(o)^k; push()}$	StackFullException

in which the notation $\mathsf{op()^k}$ is shorthand for executing $\mathsf{op()}$ in sequence k times, and $\mathsf{k} > 1$.

Regarding observer operations, there are several places that these can be tested:

(i) all observer operations could be used to observe the state after a call to any initialiser or transformer operation;

(ii) all observer operations could be used to observe the state any time a transition takes us to another state in the FSA; or

(iii) all observer operations could be used to observe the state when an FSA state is being visited for the first time.

The first of these seems like overkill, and would lead to a high number of test cases. The second is more reasonable, but considering that each FSA state represents many module states, and each of these is representative of the equivalence class to which is belong, then it seems reasonable to only test the observer operations when an FSA state is being visited for the first time. For the stack example, this gives us the following test sequences, which we instantiate into actual test cases:

Sequence	**End State**
$\mathsf{Stack(0); (isEmpty() == true); (isFull() == true); push(o)}$	StackFullException
$\mathsf{Stack(1); push(1); (isEmpty() == false); (isFull() == true);}$ $\mathsf{(top() == 1); pop(); pop()}$	StackEmptyException
$\mathsf{Stack(2); top()}$	StackEmptyException
$\mathsf{Stack(2); push(1); push(2); (isEmpty() == false); (isFull() == true);}$ $\mathsf{(top() == 2); pop(); pop(); push(1); push(2); push(3)}$	StackFullException

The additional calls to the observer operations, and the checking of their values, is equivalent to adding them into the FSA as transitions from a node to itself.

Remark 49

(i) Each of the test cases above consists of a sequence of operation calls rather than just a single call.

(ii) It is possible that a test case may consist of an equality between two sequences of operation calls, for example,

$$\mathsf{(Stack(); push(o); pop())} == \mathsf{Stack()}$$

if such a test for equality can be made.

(iii) If operations exist for forcing a module into one of the testing states then this can make testing long sequences of operation calls easier.

6.3.2 Discussion

The design of state-based systems is an important part of testing. This section has outlined that it is important to define the interface of a module to make as testable (controllable and observable) as possible. Designing for testability is as important as designing to meet requirements, even though there may be conflicts between the two.

Fortunately, many modular design notations that are part of design methodologies resembled FSAs. For example, UML uses state charts precisely for the purpose of specifying and understanding the legal sequences of operations on a class instance, and the states that result from these sequences.

Problems with State Based Testing

Of course there are some problems with FSA-based testing:

- It may take a lengthy sequence of operations to get an object into some desired state.
- FSA-based testing may not be useful if the module is designed to accept any possible sequence of operation calls. This would result in a FSA with little structure, and require a prohibitively large number of test sequences to cover.
- State control may be distributed over an entire application with operations from other modules referencing the state of the module under test.

  System-wide control makes it difficult to verify a module in isolation and requires that we identify module *hierarchies* that collaborate to achieve a particular functionality. Here the collaboration and behaviour diagrams are the most useful.

6.4 Testing Object-Oriented Programs

Object-oriented programs pose some new and interesting challenges to unit, integration and systems testing. One can view an object-oriented system can be viewed as a modular system, and therefore can test an object-oriented system by applying the techniques already discussed in this chapter. In this case, the modules are *classes*, and the operations are *methods*.

However, there are certain properties of object-oriented languages that make testing more difficult than for other imperative languages such as C. In this section, we discuss these differences and present techniques to minimise their impact.

6.4.1 Object-Oriented Programming Languages

Object-oriented programming languages support a number of features that are aimed at making the design, maintenance and reuse of code much easier. We will not go into detail but will briefly review the major features and start to think about them from a *testing* viewpoint.

The idea in object-oriented programming languages is that the language provides constructs to support encapsulation and structuring. The key elements that languages such as Java and C++ provide are:

(i) *Classes and objects* which provide the main units for structuring.

(ii) *Inheritance*, which provides the one of the key structuring mechanisms for object-oriented design and implementation.

(iii) *Dynamic Binding or Polymorphism*, which provides a way of choosing which object to use at run-time.

Classes and Objects

Classes are used as templates for creating objects. The objects do the work while the class just shows you the what sets of objects are active in the program. The common view is that a class declaration introduces a new *type* while the objects are the elements of that type. Whenever a new object is created it is an *instance*, or element, of that type. Every object in the class has:

- The *methods* defined by the class; and
- The *instance variables* (or *attributes*) defined by the class;

There can be any number of objects that are instances of a class, each with different values for their instance variables but no matter what the specific values are the object will still have the attributes and methods specified by the class. This is one thing that we can rely on in testing.

For example, the class Shape in Figure 6.3 defines a type where each object of that type contains a private hidden instance variable origin and a publicly visible method area().

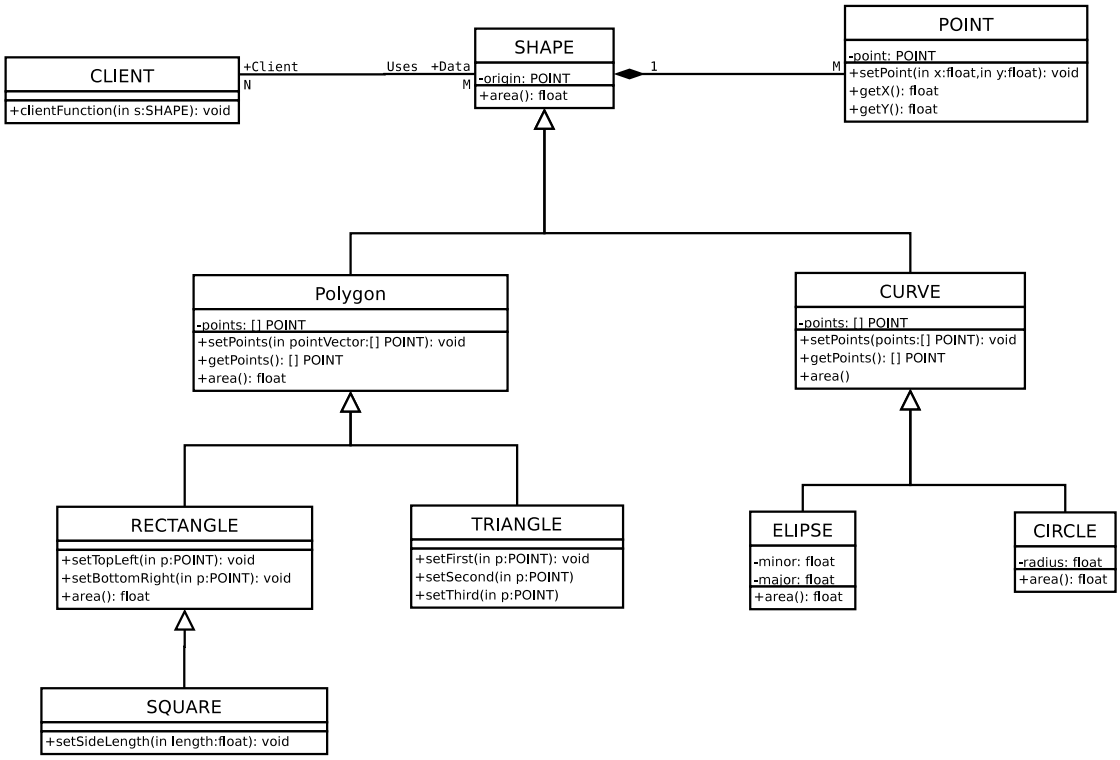


Figure 6.3: Inheritance and polymorphism.

Classes and objects do not present any major obstacles in testing on top of the encapsulation problems in testing modules. In fact, for the purpose of testing a class in isolation, one can consider an instance of a class as simply a record type, in which we have variables bound to values, and a method call to an object as simply an operation call, but with the object as an input/output value.

Inheritance

Inheritance, or *generalisation*, relationships exist between classes. The parent is the more general class providing fewer methods or more general methods while the child *specialises* the parent. A child class inherits all of the properties of its parent, but the child class may *override* operations in the parent and *extend* the parent by adding more attributes and operations.

In particular, generalisation means that the objects of the child class can be used anywhere that the objects of the parent class can be used — but not the converse. Some languages, such as Java, support only *single inheritance*, in which a class may only inherit from at most one parent class. Others, such as C++, support *multiple inheritance*, in which a class may inherit from one or more parent classes. The parent class is called the *superclass* and the child is called the *subclass*.

This idea translates to instances as well — every instance of the child class is also an instance of the parent class. Every instance the child class has the same attributes and methods as the parent — however, the child may use a different implementation of a method to that of the parent.

Inheritance is found only in object-oriented programming languages, but does not pose any immediate problems in testing. The techniques presented so far in this chapter are sufficient, however, the most immediate consequence of inheritance is that it effects the structuring of test suites.

Polymorphism

Figure 6.3 shows an inheritance hierarchy with the class Shape as the base class in the hierarchy. Any of the classes that have Shape as the superclass can be used anywhere that Shape can be used. For example, the class Client has a method called clientFunction that takes an object of type Shape as a parameter. However, the actual parameter can be an object of type Shape, an object of type Polygon, an object of type Circle or indeed an object of any type that has Shape as one of its ancestors.

This is the essence of *dynamic binding*, or as it is often called, *polymorphism* in object-oriented languages. The combination of inheritance and dynamic binding is extremely powerful for designing reusable and extendable programs. Dynamic binding and polymorphism are both found in some non-object-oriented languages, however, this is uncommon, whereas they are the essence of object-oriented languages.

The combination of inheritance and dynamic binding creates some real headaches for testers, or at least the combination of polymorphism and inheritance must be taken into account.

6.4.2 Testing with Inheritance and Polymorphism

In object-oriented testing the view is that classes are:

$$\text{class} \;=\; \text{object state} \;+\; \text{set of methods}$$

the internal states of the object become relevant to the testing. What this means in testing is that the correctness of an object depends on the internal state of the object as well as the output returned by a method call.

This is the same idea of modules being a state and a set of operations, so classes are thus the natural unit for testing object-oriented programs.

However, once this choice is made then we will need to explore the effects of object-oriented testing in the presence of inheritance/generalisation and polymorphism.

Testing Inheritance Hierarchies

Inherited features require re-testing, because every time a class inherits from its parent the state and operations of the parent are placed into new context — the context of the child. Multiple inheritance complicates this situation by increasing the number of contexts to test.

Ideally, an inheritance relationship should correspond to a problem domain specialisation, for example, from Figure 6.3 a Polygon is a special kind of Shape, a Rectangle is a special kind of Polygon and so on. The re-usability of superclass test cases depends on this idea. Unfortunately, many inheritance relationships do not respect this rule and simply inherit from classes when they want to use library functions.

Which functions must be tested in a subclass?

Figure 6.4 shows a simple parent-child inheritance hierarchy.

Parent is at the top of an inheritance hierarchy, so we derive test cases for this class first, and test it. When testing the Child class, we need to retest the number() method because this has been modified directly. However, we also need to retest the divide() method because it directly references number(), which has changed.

Can tests for a parent class be reused for a child class?

```
class Parent
{
    int number()
    {
        return 1;
    }

    float divide(int x)
    {
        return x/number();
    }
}
```

```
class Child extends Parent
{
    int number()
    {
        return 0;
    }
}
```

Figure 6.4: A simple parent child relationship for re-testing.

First we need to observe that parent.number() and child.number() are two different functions with different implementations. They may or may not have the same specification (there is no specification for this example); for example, the specification may say that the method must return a number between 0 and 10, but with no constraints as to which number — thus making the specification *non-deterministic*.

Test cases that are generated for these two different functions, however, are likely to be similar because the functions are similar. Thus, if we minimise the overloading by using principles such as the *open/closed principle* in our design, then there is a much greater chance that the inherited methods will not need new test cases in the context of the child class.

Clearly, new tests that are necessary will be those for which the *requirements* of the overridden methods change.

Building and Testing Inheritance Hierarchies

When dealing with inheritance hierarchies it is important to consider both testing and building at the same time. There are two key reasons for this:

(i) the first is to keep control of the number of test cases and test harnesses that need to be written; and

(ii) the second is to make sure that we know where faults occur within the inheritance hierarchy as much as possible.

A first approach to inheritance testing involves *flattening the inheritance hierarchy*. In effect each subclass is tested as if all inherited features were newly defined in the class under test — so we make absolutely no assumptions about the methods and attributes inherited from parent classes. Test cases for parent classes may be re-used after analysis but many test cases will be redundant and many test cases will need to be newly defined.

A second approach to testing and building is *incremental inheritance-based testing*. Incremental building and testing proceeds as follows:

Step 1 First test each base class by:

(1) Testing each method using a suitable test case selection technique; and

(2) Testing interactions among methods by using the techniques discussed earlier in this chapter.

Step 2 Then, consider all sub-classes that inherit or use (via composition or association) only those classes that have already been tested.

A child inherits the parent's test suite which is used as a basis for test planning. We only develop new test cases for those entities that are directly, or indirectly, changed.

Incremental inheritance-based testing does reduce the size of test suites, but there is an overhead in the analysis of what tests need to be changed. It certainly reduces the number of test cases that need to be selected over a

flattened hierarchy. If the test suite is structured correctly, test cases can be reused via inheritance as well, which provides the same benefits for conceptualisation and maintenance for test suite implementations as it does for product implementations.

Inheritance-based testing can also be considered to be a form of regression testing where the aim is to minimise the number of test cases needed to test a class modified by inheritance.

Implications of Polymorphism

Consider the inheritance hierarchy in Figure 6.3. The implementation of area() that actually gets called will depend on the state of the client object and the runtime environment.

In procedural programming, procedure calls are *statically* bound — we know exactly what function will be called at compile time — and further, the implementation of functions do not change (well, not unless there is some particularly perverse programming) at runtime.

In the case of object-oriented programming, each possible binding of a polymorphic class requires a separate set of test cases. The problem for testing is to find all such bindings – after-all the exact binding used in a particular object instance will only be known at run-time.

Dynamic binding also complicates integration planning. Many service and library classes may need to be built and tested before they can be integrated to test a client class.

There are a number of possible approaches to handling the explosion of possible bindings for a variable. One approach is to try and determine the number of possible bindings through a combination of static and dynamic program analysis.

Consider the Client class in Figure 6.3. The problem from a testing point of view is to know which actual Shape object gets called at run-time. If an instance of Square is bound to the variable s in clientFunction then we would expect a different result for the area() computation than if that instance Circle was bound to s.

If we instrumented the code for Shape and all of its descendants to reveal the types of the actual objects that are bound to S then we could use that information to determine the subset of the class hierarchy that is actually used by the method clientFunction. The subset can then be covered more thoroughly with test cases. For example, it may be the case that Polygon hierarchy is used by the client function while Curve hierarchy is not. If that is the case the testing clientFunction and the classes in the Polygon hierarchy makes sense but testing the classes in the curve hierarchy makes less sense.

Beware however, this approach is not foolproof and is biased heavily towards the data used to generate the bindings.

Chapter 7

Software Reliability Theory and Practice

7.1 An Introduction to Software Reliability

So far we have looked at testing methods based that are aimed at finding the faults in programs. This leaves open some big questions such as when to stop testing and what sort of assurances can you give when you do stop testing.

More on that later. For now, we will begin this chapter by taking a different view of testing; one that is more consistent with system reliability and its measurement.

We have already made the point that software is part of a much larger *system* consisting of hardware and communications as well as programs. Often one is required to build a system that is far more *reliable* than any one of its components. For example, safety standards and high integrity systems standards often demand the reliability of systems to have a failure probability in the order of 10^{-9}(but we will talk at some length about numbers later). This means a failure every 114,000 years on average!

Similar kinds of numbers are required for many commercial systems as well, such as web-servers. For example, how reliable/available must a web-server be to handle the number of transactions the site receives? In fact, software is invading many new areas. With these areas comes a demand for reliability that must be quantifiable and often justifiable. Examples of such domains include, but are not limited to:

- **The automotive industry** — electronics and computers now feature in many new cars and include systems such as ABS, Traction Control, Electronic Stability Control, or Engine Management Systems.
- **The avionics (aeroplanes) industry** — where *fly-by-wire*, that is flying planes by computer is becoming common-place.
- **Process control industries** — such as the food and beverage industry, or chemical plants, where automation is playing a larger role. Here robots and computer systems need to be operating for long lengths of time without maintenance.
- **The Internet** — where web-sites or ISPs hosting data ware-houses and providing other services need to be operational for long periods of time in in order to provide high qualities of service.

Software reliability can be used in a number of ways in development.

Evaluation of New Systems: Software reliability can be used to evaluate systems by *testing* the system and gathering reliability measures. The measures can be useful in deciding what third party software, such as libraries, to use in a project.

Evaluation of Development Status: Software reliability can be used to evaluate development status during testing. This is often done by comparing current failure rates with desired failure rates. Large discrepancies may indicate the need for action.

Monitor the Operational Profile: Software reliability can be used to monitor the operational profile of systems and evaluate the effects of changes or new features as they they are added to the system.

Understanding Quality Attributes for the System: Provided that we adopt appropriate definitions of a *failure* then software reliability can be used to explore quality attributes of a project and the factors affecting it by functional testing.

Viewed from a certain philosophical perspective, reliability is interesting not for its theory and methods, but what it says about programs. So far, we have conducted a meticulous analysis of input/output domains, the specifications of units and components in order to derive test cases that will detect systematically detect failures.

In reliability growth models, the aim is to select tens of thousands of test inputs and see how many of them *fail*. In reliability assessment and modelling, we treat components as black boxes and perform *experiments* on them to assess their reliability.

In this chapter we will look at the following topics to gain an appreciation of the scope and methods for software reliability.

(i) **Software Reliability Engineering Methods**

(ii) **Random Testing**

(iii) **Reliability Evaluation Methods**

(iv) **Reliability Growth Models**

The aim is for you to have a good appreciation of the ideas behind reliability, how to distinguish between system and software reliability, and to gain an appreciation of what reliability engineering can contribute to a project.

Along the way we will need to review our basic probability theory, stochastic processes and Markov models. The mathematical theory serves to justify the results that we will present but in practice we can often ignore the mathematical theory provided we understand how to interpret the results.

7.2 What is Reliability?

Reliability is one of a number of dependability attributes of a program. Dependability is usually analysed in terms of:

The attributes of dependability: are not like the attributes of quality where dependability is broken down into a set of measurable attributes. Rather, it is a collective name for a set of attributes that have been discussed in the literature for a number of years. The attributes themselves are shown in Figure 7.1.

The means for achieving dependability: are approaches, techniques and methods, for improving the outcomes for a particular attribute. For example, *Fault Tolerance* is the means for improving reliability.

The threats to dependability: are the things act against the attribute, for example, faults actively lower system reliability.

The taxonomy for dependability attributes, the means for achieving it and the threats to it are given in Figure 7.1.

In the specific case of reliability, we have the following definition.

> **Definition 50** Reliability is the probability that a system will operate without failure for a specified time in a specified environment.

We need to look into this definition in some more detail. To see how it works, consider the two simple systems consisting of two writers and a reader communicating via a shared memory on a bus in Figure 7.2.

Suppose that the probability of failure of each channel of the bus is 10^{-5} per operational hour. For the single channel bus therefore, the probability of a bus failure, such a message becoming corrupt or the bus failing to

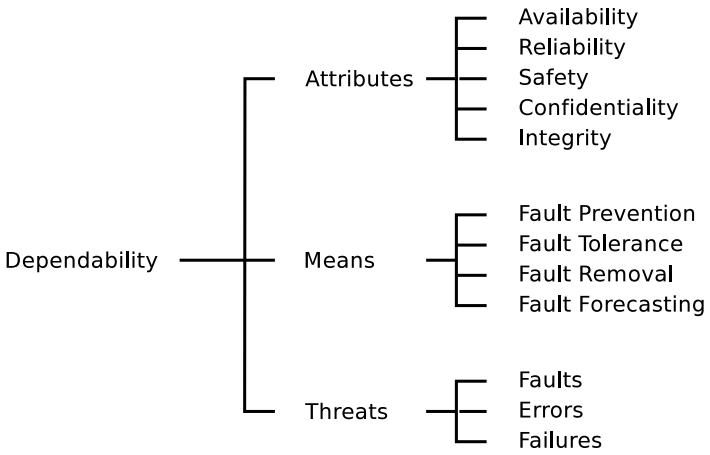
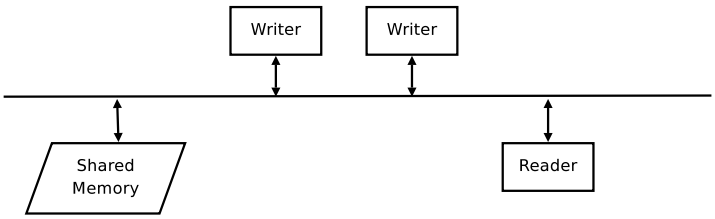
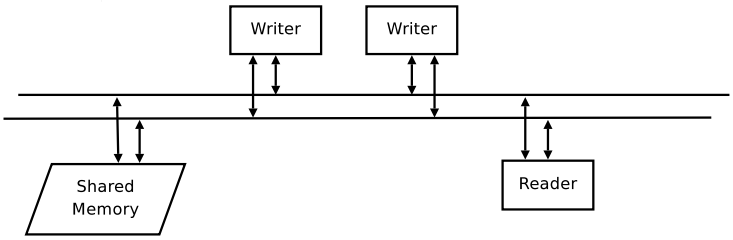


Figure 7.1: The dependability attributes of a system (not just a program) according to Laprie.



A reader and two writers communicating through a shared memory on single channel bus.



A reader and two writers communicating through a shared memory on dual channel bus.

Figure 7.2: Two bus architectures for communicating between reader and writer.

transmit a message, is 10^{-5} per operational hour, or one failure every 11.4 years. If we add the second channel then the failure probability decreases to 10^{-10}, or one failure in every 1,114,552 years, because both channels must fail at the same time for this to occur.

We have to be careful in not over interpreting these numbers. While the improvement is significant it is, after all, only a *probability*.

Note that the definition of reliability that we have used depends heavily on determining, and in practice detecting, failures.

Definition 51 *A* failure *is the departure of the results of a program from requirements.*

Remark 52 The definition of failure is extremely general and can include performance failures such as excessive response times. Also note that the definition of failure is a *dynamic concept*: we must execute the program for a failure to occur. Intuitively, the ways in which a program may fail are by:

(i) activating a single fault, or even a series of faults, within the program itself that results in a failure; or

(ii) interacting with the environment in such a way that the interaction causes a failure of the program.

Such failures are typically caused by design faults.

In their book on software reliability, Musa *et al.* [1] define reliability as follows:

> "*Software reliability concerns itself with how well the software functions to meet the customer requirements. . . . [Reliability] is the probability that the software will work without failure for a specified period of time.*"

Thus, we think of reliability as a *dynamic* concept that relates to an executing program. You can contrast Definition 50 with the more *static* concept of defect counts that try and determine quality from the software design or other statically determined artifacts.

Classifying Failures

Our definition of reliability depends quite critically on the interpretation of *failures*. Examples of failures include:

(i) *functional failures* where the actual behaviour of the system deviates from the specified functional behaviour;

(ii) *timing failures* where the program may deliver correct results but the program fails to meet its timing requirements;

(iii) *safety failures* where an accident resulting in harm or injury is deemed to be a failure of the system;

In practice we will need a way of *observing* failures and logging each failure as it arises. We also need a way of determining the time at which a failure occurs, called the *failure time*, or the rate at which failures occur, called the *failure rate*. From the failure times or the failure rates we can estimate a system's reliability. We will also need assurance that the estimates are accurate with respect to the program we are measuring.

The definition also implies that we should understand the environment in which the system must operate and that the reliability of the system may, and often will, be effected if it is used in different environments.

We can discern a number of different types of failure and our program, as well as our methods for observing failures, must account for them.

Transient Failure – In this case the program gives and incorrect result, but the program continues to execute.

Hard Failure – In this case the program crashes (stack overrun, heap overrun, broken thread).

Cascaded Failure – In this case the program crashes and takes down other programs.

Catastrophic Failure – In this case the program crashes and takes down the operating system or the entire system; a total system failure

Software Reliability Engineering

Software reliability engineering is our first real engineering method. The general phases and activities of SRE are set out in Table 7.1.

Phase	**SRE Activities**
Requirements	Determine the functional profile of the system. Determine and classify important failures. Identify client reliability needs. Conduct trade-off studies. Set reliability objectives.
Design	Design and evaluate systems to meet reliability goals. Allocate reliability targets to components. (Reliability Evaluation Methods here!)
Implementation	Measure and monitor the reliability during implementation. Manage fault introduction and propagation.
System and Field Testing	Measure reliability growth (using reliability growth models). Track testing progress against reliability growth.
Post-delivery and Maintenance	Monitor field reliability against reliability objectives.

Table 7.1: The phases and activities for Software Reliability Engineering.

We can make some general observations that apply to many software engineering methods.

- The activities of software reliability engineering span the entire software development process but the activities themselves do not constitute a development process.
- The activities themselves need to be supported by theoretical tools for the analysis steps. The analytical tools take the form of *reliability models* and evaluations methods such as *Markov models*.
- Further, design and implementation of reliable systems is supported by design principles and algorithms such as those that we take from the discipline of *fault tolerant systems*.

In the remainder of this chapter we will begin by taking a brief look at reliability block models, and then at Markov models as a model of reliability. From Markov models we proceed to the reliability growth models of Musa *et al.* [1]. In the final section of this chapter, we discuss error seeding as a means of measuring reliability.

7.3 Random Testing

As one can derive from the name, in random testing, the test inputs are chosen at random. The main applications of random testing are reliability measurement and security testing (Chapter 8), because it gives a good statistical

spread of the inputs to a program. Random testing has also been used with some success for testing functional correctness.

Random testing does not mean abandoning the analysis of the input and output domain and the word "random" does not mean that the test cases are chosen completely *arbitrarily*. What it means is that the test inputs are chosen with a particular distribution over the input domain that somehow reflects the actual usage of the program.

For random testing the distribution of inputs comes from an *operational profile* for the program. The operational profile of a program is the probability distribution of its input domain when the program is used in actual operation. Put another way it is the probability that an element in the input domain will be chosen if the program is actually being used.

Figure 7.3 shows a use case diagram, taken from 4, for an order handling system. The diagram contains six base cases, three inclusions, and three extensions.

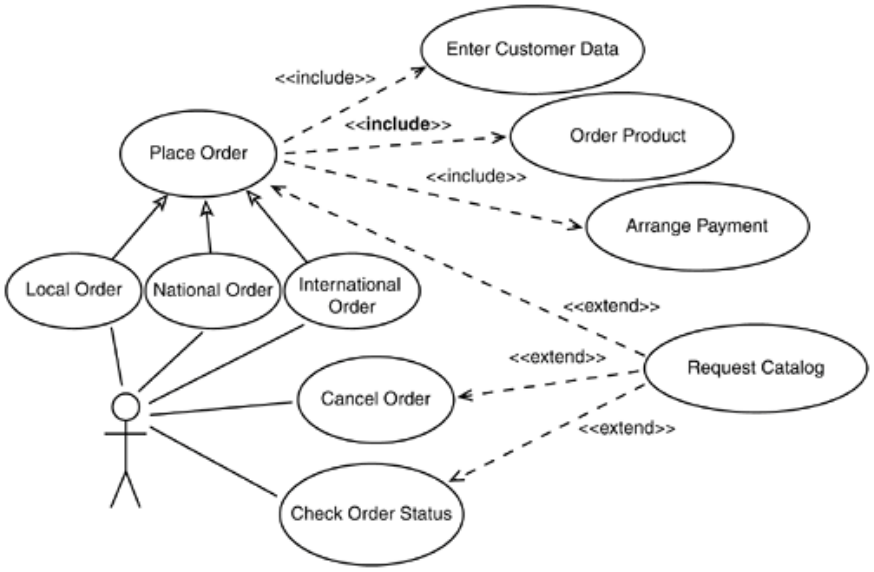


Figure 7.3: A use case diagram for an order handling system

Figure 7.4 shows an operational profile for the use case model from Figure 7.3. Clearly, this was derived with additional information not presented here, so this operational profile is for illustrative purposed only. A cell in the matrix estimates the probability that the included or extended use case is invoked for the given base case; e.g. when a local order is placed, 80% of the time, the customer details must be entered.

In these notes, we do not cover how to generate operational profiles, however, clearly there are several ways, including: looking at historical data; estimation based on personal usage; surveys; and looking at probability distributions in similar systems.

In random testing, test inputs are randomly generated according to the operational profile[1], and data, such as failure times, are recorded when a random sample of test cases are executed. The data obtained from random testing is then be used to estimate reliability (see Chapter 7). No other testing method can be used in this way to estimate reliability. However, we need to take care with operational profiles – the distribution of inputs taken from an operational profile will effect reliability predictions.

To generate a random test input for an operational profile, *Place Local Order* would be selected with a probability of 3%, and then *Enter Customer Details* would be selected with a probability of 80% of these times (so a total probability of 2.4% of the time).

[1]For a discussion of good random number generator, see, e.g., D.E. Knuth, *The Art of Computer Programming, vol. 2: Semi-numerical Algorithms*, 2nd Ed., Addison Wesley, 1981.

Base Use Cases	Times per Day	Probability
Place Order	0	0.00
Place Local Order	50	0.03
Place National Order	350	0.20
Place International Order	25	0.01
Cancel Order	25	0.01
Check Order Status	100	0.06

Include and Extend Use Cases	Times per Day Standalone	Times per Day with Base Use Case	Times per Day	Probability
Enter Customer Data	0	340	340	0.20
Order Product	0	340	340	0.20
Arrange Payment	15	340	355	0.21
Request Catalog	40	89	129	0.08
TOTAL			1714	1.00

Probability of Use by Base Use Case			
Enter Customer Data	Order Product	Arrange Payment	Request Catalog
Include	Include	Include	Extend
0.80	0.80	0.80	0.15
0.80	0.80	0.80	0.15
0.80	0.80	0.80	0.15
0.80	0.80	0.80	0.15
			0.20
			0.20

Figure 7.4: An example operational profile for the system described in Figure 7.3

7.4 Reliability Block Diagrams

What is clear from the reliability engineering process in Table 7.1 is that we need some way of predicting and evaluating the reliability of our designs. For systems there are well established modelling methods:

(i) Reliability block diagrams; and

(ii) Markov models.

These models are *stochastic* in nature. A stochastic process is one in which there is a sequence of events in time where each event is part of a probability distribution.

Series—Parallel Systems

Reliability block diagrams seek to decompose systems into parallel and serial blocks where each block interacts with other blocks to effect the reliability of the system as a whole.

Consider again the example of the dual channel bus in Figure 7.5 where a redundant pair of writers put values from the source onto the bus, and a reader that reads the values from the bus. The system fails if no message is received by the reader in the given time interval.

If we consider a message starting at the source then it is passed to both writers and both writers pass the message to both channels. We might model this situation as in Figure 7.6.

In the model, each writer acts in serial with the bus and both

channels of the bus act in serial with the reader. Thus, to analyse this situation we must calculate the reliabilities for the *serial* and *parallel* blocks in the model.

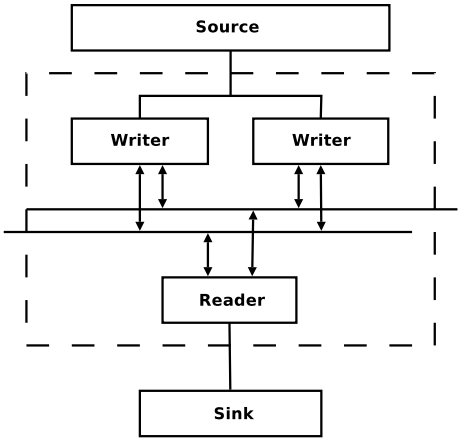


Figure 7.5: The dual channel bus.

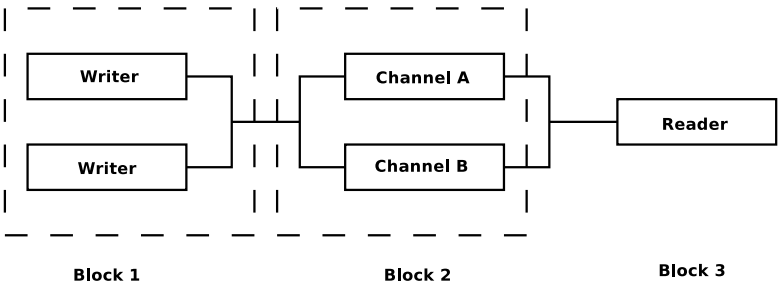


Figure 7.6: A reliability block model for the readers and writers in Figure 7.5.

Reliability and Failure Logic for Serial Blocks

Consider the serial system in Figure 7.7. For this system to function without failure at a given time T, each of the components B_1, B_2, and B_3 must function independently without failure for the time period T. Therefore, for the system to fail in the time T either **Block 1** must not fail in time T, **Block 2** must not fail in time T, and **Block 3** must not fail in time T.

If the reliability of each block is independent of the other blocks then a direct application of the law of conditional probability yields

$$P\{B_1 \text{ functions } \wedge \ B_2 \text{ functions } \wedge \ B_3 \text{ functions}\} = R(B_1)R(B_2)R(B_3)$$

where $R(B_i)$ denotes the probability of **Block i** functioning without failure for time T. Put another way, $R(B_i)$ is the probability that the first failure occurs *after* time T, or not at all. $R(B_i)$ is the reliability of **Block i**.

Definition 53 In general, for systems in series, $R(B) = \prod_n R(B_i)$ using reliability logic.

Failure logic is the converse to reliability logic. For the system to fail in the time period T only one of the components needs to fail. Thus the probability of of failure is

$$P\{B_1 \text{ fails } \vee \ B_2 \text{ fails } \vee \ B_3 \text{ fails}\}.$$

The law of additive law of probability gives us:

$$\begin{aligned} F(B) &= F(B_1) + F(B_2) + F(B_3) - F(B_1)F(B_2) - F(B_1)F(B_3) - F(B2)F(B_3) \\ &+ F(B_1)F(B_2)F(B_3) \end{aligned}$$

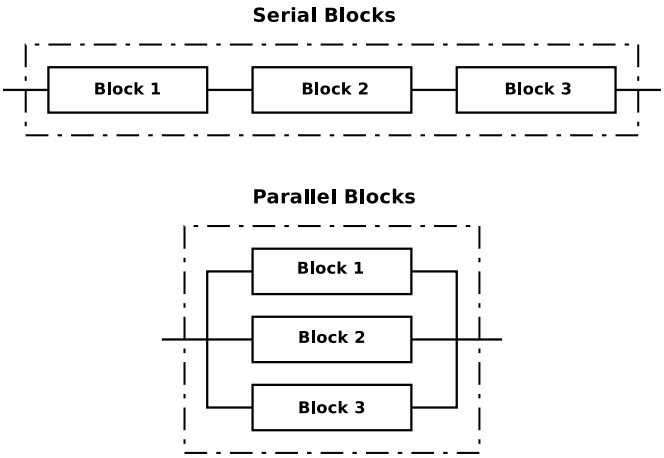


Figure 7.7: The serial composition of three blocks.

where $F(B_i)$ is the probability that the first failure in B_i will occur before time T. This equation says that the probability of the parallel block failing is the probability of any of the blocks failing independently ($F(B_1) + F(B_2) + F(B_3)$), minus the chance that any of the blocks fail simultaneously (because one failing will mean that all fail).

Definition 54 In general, for systems in series, $F(B) = 1 - \prod_n (1 - F(B_i))$ using failure logic.

Reliability Logic for Parallel Blocks

For parallel systems *all* blocks must fail before the system fails and only one block needs to function for the system to function. Parallel blocks, it turns out, work almost inversely to serial blocks.

For a parallel block above to fail in a time T we need all of the blocks to fail within the time T and so

$$F(B) = P\{B_1 \text{ fails } \wedge \ B_2 \text{ fails } \wedge \ B_3 \text{ fails}\} = F(B_1)F(B_2)F(B_3)$$

provided that B_1, B_2 and B_3 fail independently.

Definition 55 In general for a parallel system to fail $F(B) = \prod_n F(B_i)$ using failure logic.

For parallel blocks to be reliable for a time T we have must experience no failure in time T and therefore, to be reliable, a parallel block only requires that a single block is functioning. This gives us the following formula for reliability.

$$\begin{aligned} R(B) &= P\{B_1 \text{ functions } \vee \ B_2 \text{ functions } \vee \ B_3 \text{ functions}\} \\ &= R(B_1) + R(B_2) + R(B_3) - R(B_1)R(B_2) - R(B_1)R(B_3) - R(B_2)R(B_3) \\ &+ R(B_1)R(B_2)R(B_3). \end{aligned}$$

This comes from the additive law of probabilities.

Definition 56 In general for parallel blocks we have $R(B) = 1 - \prod_n (1 - R(B_i))$ using reliability logic.

The key relationships between failure logic and reliability (or success) logic lay in the equation

$$R(B) = 1 - F(B) \tag{7.1}$$

where

$$R(B) = P\{\text{First Failure at time } \tau > T\}$$

and

$$F(B) = P\{\text{First Failure at time } \tau < T\}.$$

Creating Reliability Block Diagrams from System Architectures

Here are some general guidelines for creating reliability block models from system architecture, especially if presented as a set of function block diagrams.

Guideline 1 Determine what constitutes a *system* failure. This in turn determines which component failures cause a system to fail.

Guideline 2 Determine which components need to fail in order to cause a system failure. Some guidelines for constructing reliability block models is to consider how messages, signals or data flows through the system and the consequence of what happens if these are corrupted or interrupted.

Guideline 3 Try to ensure that each block in the reliability block model captures one function of the system.

Guideline 4 Try to ensure that you capture the parallel/serial connections from the system accurately in your reliability block model.

Guideline 5 There may be more than one mode for the system — you will need to create a reliability block diagram for each mode of the system. For example, Windows operating systems can operate in a *safe mode*, which is more reliable because it is limited to basic functionality.

Back to the Reader/Writer Example

Each writer acts independently of the other writer and then communicates via the bus to the reader. So now we use the guidelines above to get the reliability block model in Figure 7.8. Suppose we measure the failure probabilities

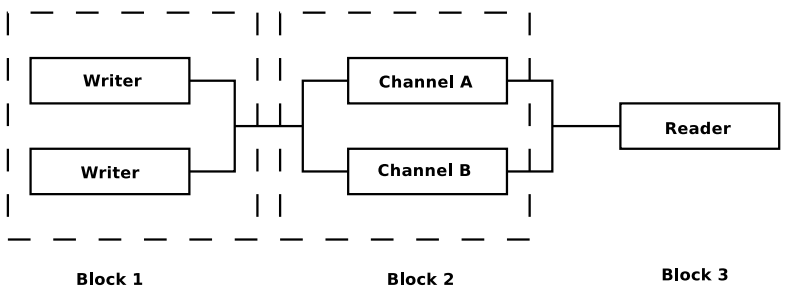


Figure 7.8: The reader/writer example revisited.

by and arrive at the estimates in Table 7.2.

Component	**Failure Probability in Time** T
Writers	$F(W) = 10^{-4}$
Bus Channels	$F(B) = 10^{-2}$
Reader	$F(R) = 10^{-4}$

Table 7.2: Failure times for the reader/writer in Figure 7.8.

Firstly, we have only failure probabilities but we are after reliability figures. So, note that the reliability of a writer is $R(W) = 1 - F(W)$. Now, using reliability logic for parallel blocks as in Block 1 we calculate the reliability of

two writers as follows.

$$\begin{aligned} R(B_1) &= 1-(1-R(W))(1-R(W)) \\ &= 1-(1-(1-F(W)))(1-(1-F(W))) \\ &= 1-F(W)^2 \end{aligned}$$

With the square deriving from the fact that there are two writers in that block, so the probability of the block failing is the product of the probabilities of the either writer failing.

For Block 2 we likewise calculate the reliability of two channels in parallel:

$$\begin{aligned} R(B_2) &= 1-(1-(1-F(B)))(1-(1-F(B))) \\ &= 1-F(B)^2 \end{aligned}$$

Finally we have a system in series so to calculate the reliability of the system we need to apply reliability logic again:

$$R(System) = R(B_1)R(B_2)R(B_3)$$

which gives:

$$\begin{aligned} R(System) &= (1-F(W)^2)(1-F(B)^2)(1-F(R)) \\ &= (1-10^{-8})(1-10^{-4})(1-10^{-4})) \\ &\approx 0.998 \end{aligned}$$

The system reliability is the probability of failure free operation over a fixed time period T. It does not tell us how reliability changes over time. Determining how the reliability of a system changes over time is a stochastic property.

7.5 Markov Models

Reliability block diagrams are useful for analysing failure probabilities and reliability over fixed time intervals but there are many occasions when we need to understand how reliability changes over time; especially in the presence of hardware. This is where a *stochastic process* such as a Markov Model can provide extra information about how a system's reliability changes over time.

To understand stochastic processes you will need to be familiar with probability theory. There is a brief review of the key ideas behind probability in Appendix A.

Stochastic Processes

Random variables are functions that assign a number to each outcome in the sample space of an experiment. See Appendix A for more information about random variables. The random variables that are typically encountered in reliability models are:

(i) the number of failures at time T which is *discrete*, or an integer-valued, random variable; or

(ii) the time at which the first failure occurs, which is a *continuous*, or real-valued, random variable.

A stochastic process is a sequence of such random variables $\{\xi(1), \ \ldots, \xi(k)\}$ with the same underlying sample space.

The set of all possible values, $\xi(T)$, for the random variable is called the *state space* for the system. The value of $\xi(T)$ at time T is the state of the system. Changes of state are called *transitions*.

Now, if we look at the sequence of random variables $\{\xi(1), \ldots, \xi(k)\}$ again then what we see is a sequence of states chosen at random from the state space S. For example, we can have a random variable $\xi(T)$ that returns the number of failures experienced by time T. If we consider time intervals

$$T = \tau_1,\ \tau_2,\ \ldots, \tau_k$$

then we have a stochastic process which is given by

$$\xi(\tau_1),\ \xi(\tau_2),\ \ldots,\ \xi(\tau_k)$$

with each $\xi(\tau)$ returning the number of failures experienced by time τ.

The *realisation* of the stochastic process is the sequence of actual values

$$\xi(0),\ \xi(1),\ \xi(2),\ \ldots,\ \xi(k)$$

where each $\xi(T)$ is the actual number of failures experienced by time T and can be written out as a table as follows.

Time	**Number of Failures Experienced**
0	0
1	6
2	11
3	14
4	17
5	18
⋮	⋮

A Motivating Example

The reason we are interested in Markov systems is that we want to explore how the reliability of systems evolves over time. At the heart of most of our reliability modelling will be the simple finite state automaton (FSA) in Figure 7.9.

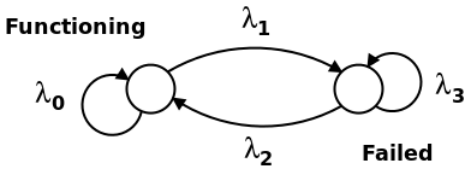


Figure 7.9: The two state model — functioning and failed.

There are two states of interest in the automaton - the *functioning* state and the *failed* state. The transitions in the automaton are not labelled with events but are labelled with probabilities. Each λ_i is the probability of making a transition from one state to the next *within a specified time interval*.

In particular, for the simple model of failure processes in Figure 7.9, we can interpret the states and probabilities over a time interval Δt as in Table 7.3. Of course, to make sense $\lambda_0 = 1 - \lambda_1$ and $\lambda_3 = 1 - \lambda_2$ so that the sum of the probabilities on all of the transitions leaving a state is 1. The probabilities are called the *transition probabilities* for the system.

The Transition Matrix for the Two State Model

If we look at the probabilities for the simple two model as in Table 7.4 then the probabilities form a matrix. We

λ_0	The system remains in a functioning state over Δt.
λ_1	The system fails during Δt.
λ_2	The system is repaired over Δt.
λ_3	The system remains in a faulty state over Δt.

Table 7.3: Interpreting the states and events of the automaton in Figure 7.9.

	Functioning	**Failed**
Functioning	$1-\lambda_1$	λ_1
Failed	λ_2	$1-\lambda_2$

Table 7.4: The transition matrix for the two state model.

read the matrix along its rows. The first row tells us that there is a probability of $1-\lambda_1$ of making a transition from a functioning state to a functioning state and a probability of λ_1 of making a transition from a functioning state to a failed state. This matrix is called the *transition matrix* matrix for the Markov Model and is usually just written in matrix form:

$$T = \begin{bmatrix} 1-\lambda_1 & \lambda_1 \\ \lambda_2 & 1-\lambda_2 \end{bmatrix}$$

when the interpretation is clear.

Analysis of the Simple Two State Model

Suppose now that $\Delta t = 1$ second and the system that the FSA in Figure 7.9 models is a communications bus. *What state are we likely to be in after 1 second*?

What we have is are the following conditional probabilities:

$$\begin{aligned} P\{\text{Functioning}\,|\,\text{Functioning}\} &= 1-\lambda_1 \\ P\{\text{Failed}\,|\,\text{Functioning}\} &= \lambda_1 \\ P\{\text{Functioning}\,|\,\text{Failed}\} &= \lambda_2 \\ P\{\text{Failed}\,|\,\text{Failed}\} &= 1-\lambda_2 \end{aligned}$$

which are just obtained from the transition matrix. So, the probability of being in the functioning state and making a transition to the functioning state is $1-\lambda_1$.

What state will the bus be in after two seconds?

This is the sum of the conditional probability

$$P\{\text{Functioning}\,|\,\text{Functioning at time } T=1\}$$

and

$$P\{\text{Functioning}\,|\,\text{Failed at time } T=1\}.$$

If the system starts at $T=0$ in the functioning state then the probability that it is in the functioning state at time $T=1$ is $1-\lambda_1$. The probability that the system is in a failed state at time $T=1$ is λ_1 if it starts in the functioning state.

Let $S(T)$ be the state of the system at time T. Then

$$\begin{aligned} P\{S(2) = Functioning\} &= P\{S(2) = Functioning \,|\, S(1) = Functioning\} \\ &+ P\{S(2) = Functioning \,|\, S(1) = Failed\} \end{aligned}$$

We know that $P\{S(0) = Functioning\}$. Now, using the transition probabilities, from Figure 7.9 and the assumption that the transitions are independent of one another we get:

$$P\{S(2) = Functioning\} = (1-\lambda_1)(1-\lambda_1) + \lambda_1\lambda_2$$

The probabilities can be calculated much quicker by using the transition matrix:

$$\begin{bmatrix} 1-\lambda_1 & \lambda_1 \\ \lambda_2 & 1-\lambda_2 \end{bmatrix}.$$

Now, apply the assumption that the transitions are independent of the the previous states of the system; we just look at the current state. Therefore:

$$\begin{bmatrix} 1-\lambda_1 & \lambda_1 \\ \lambda_2 & 1-\lambda_2 \end{bmatrix} \times \begin{bmatrix} 1-\lambda_1 & \lambda_1 \\ \lambda_2 & 1-\lambda_2 \end{bmatrix} = \begin{bmatrix} (1-\lambda_1)(1-\lambda_1)+\lambda_1\lambda_2 & (1-\lambda_1)\lambda_1+\lambda_1(1-\lambda_2) \\ \lambda_2(1-\lambda_1)+(1-\lambda_2)\lambda_2 & \lambda_2\lambda_1+(1-\lambda_2)(1-\lambda_2) \end{bmatrix}$$

Of course the resulting state is just:

$$\begin{bmatrix} 1-\lambda_1 & \lambda_1 \\ \lambda_2 & 1-\lambda_2 \end{bmatrix} \times \begin{bmatrix} 1-\lambda_1 & \lambda_1 \\ \lambda_2 & 1-\lambda_2 \end{bmatrix}$$

when the matrices are multiplied out.

The sequence of states visited by the process over a k-second period gives *one possible* path that the process can take. This is one chain that the sequence of random variables that the process can take.

Markov Models

> **Definition 57** A *Markov model* (or *Markov chain*) is a stochastic process with the *Markov property*. The Markov property means that, at any state of the process, the future states of the process are dependent only on that state.

The above definition states that all future states of the process depend only on the current state of the process and not on any states of the system prior to the current state. Formally:

$$\begin{aligned} & P\{\xi(n+1) = k \,|\, \xi(n) = j_n,\ \xi(n-1) = j_{n-1}, \ldots, \xi(1) = j_1\} \\ = \ & P\{\xi(n+1) = k \,|\, \xi(n) = j_n\} \end{aligned}$$

Put another way, the probability that we end up in state k given that the process has been in states j_n, j_{n-1} …, j_1 depends only on the fact that we are in state j_n and not on the fact that the system had been in any of the other states.

The system specified in Figure 7.9 is a Markov model because it is modelled using a FSA, and FSAs have the property that future states are determined by the current state. Although FSAs are not necessarily stochastic, they can be used to specify stochastic processes by simply adding probabilities to each of the transitions, as is done in Figure 7.9.

Another useful distinction is between homogeneous and in-homogeneous Markov models.

> **Definition 58** If the probabilities in the transition matrix are fixed and do not vary with time then the process is called *homogeneous*. Otherwise it is called *in-homogeneous*.

In general, if we wish to calculate the state of an homogeneous Markov Process with transition matrix M after n steps then the result is given by calculating:

$$M^n,$$

in which M^n is defined as $M \times M \times \ldots \times M$ n number of times. M^n is called the ***n step transition matrix*** for the Markov model.

To calculate the probability distribution after n steps, we need to supply the transition matrix with an *initial probability distribution*. A *probability distribution*[2] is a list of the states together with their probabilities. The initial distribution is thus just a vector of probabilities that specify the probability of starting in each of the system states.

If ***M*** is an $m \times m$ matrix then then the initial probability distribution is given by a vector α of length m. The probability distribution of the system after time n is thus:

$$\alpha M^n \tag{7.2}$$

For example suppose that

$$M = \begin{bmatrix} 0.9500 & 0.0500 \\ 0.0200 & 0.9800 \end{bmatrix}$$

and we compute $M^{200}\alpha$. The transition probabilities are shown graphically in Figure 7.10, with the length of the vector on the x axis, and the probability on the y axis. The transition probabilities are derived by plotting the four transition probabilities of the matrix λ_{11}, λ_{12}, λ_{21} and λ_{22} over time. The positioning of the boxes is relative to their cell in the above matrix.

Now suppose that we change the probability of failure to 10^{-4}:

$$M = \begin{bmatrix} 0.9999 & 0.0001 \\ 0.0200 & 0.9800 \end{bmatrix}$$

and again compute M^{200}. The graphs for the four transition probabilities of the matrix λ_{11}, λ_{12}, λ_{21} and λ_{22} now change to the graphs shown in Figure 7.11.

Remark 59 Markov models are often the method of choice for analysing combined software and hardware systems. They offer a more dynamic analysis than reliability block diagrams.

(i) With Markov models we can postulate different scenarios and try "what-if" experiments relatively quickly; for example, we can ask what if we had different transition probabilities, or what if we had different initial probability distributions?

(ii) It is also possible to vary the time bounds on our experiments, graph the reliabilities over time and explore how the reliability of the system changes over time. The main drawback is the problem of state explosion when the number of components becomes large.

A Second Example

As a second example, consider a system that consists of three components connected as in Figure 7.12.

If we want to study the failure probabilities of the system over time then our analysis will typically take the following steps.

Step 1 — Determine the *Markov state* for the system. The Markov state needs to be distinguished from the system state or indeed the state of any individual component.

Step 2 – Calculate the *transition probabilities* between Markov states and consequently derive the *transition matrix* for the system.

[2] See Appendix A for a definition of probability distributions.

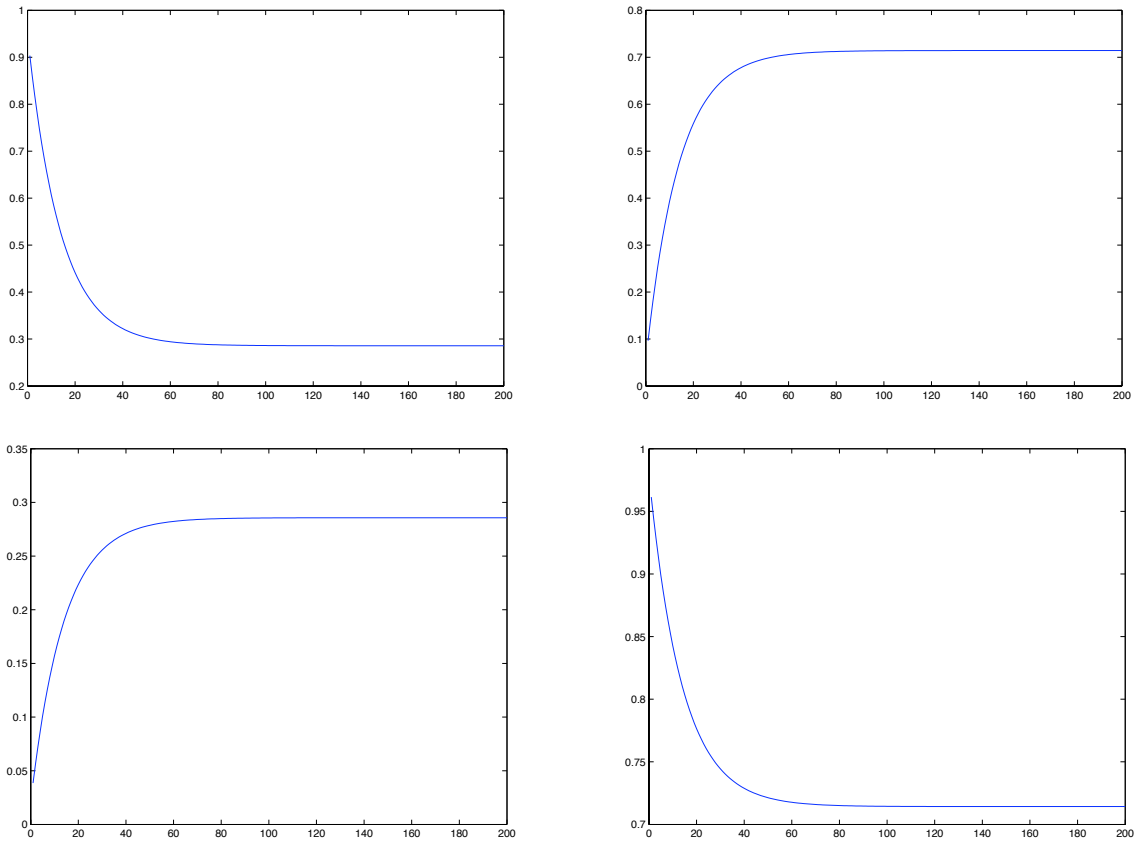


Figure 7.10: The transition probabilities after 200 seconds.

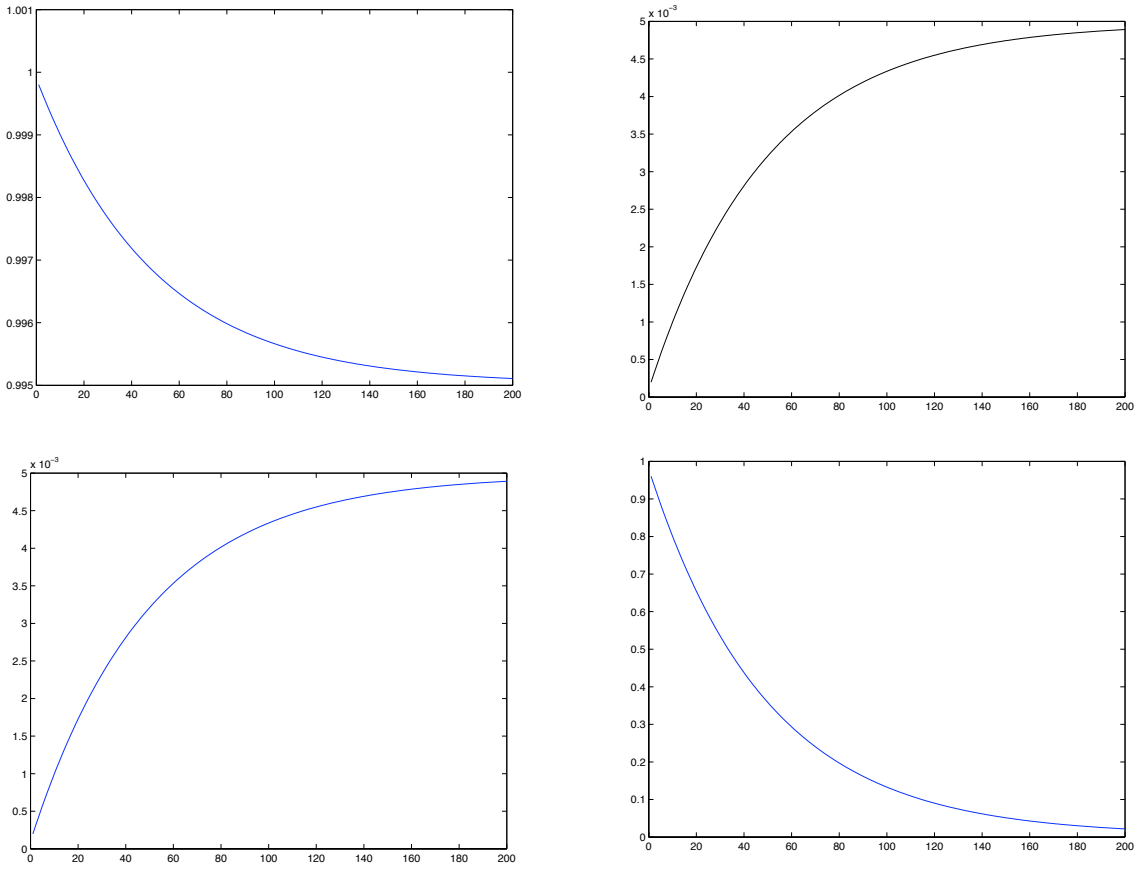


Figure 7.11: The graphs for the transition probabilities after being modified.

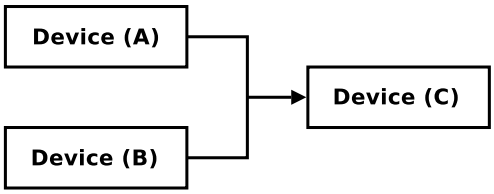


Figure 7.12: Three device system.

Step 3 – Determine the time frame of interest and calculate the n-step transition probabilities.

Each of the devices can be in either a *functioning* state or a *failed* state. The first step is to construct a composite Markov state for the system based on the states of the individual devices.

Let A stand for the fact that a device is *alive*, or *functioning*, and F for the fact that a device has *failed*. Let S_A be the state of device A, S_B be the state of device B, and S_C be the state of device C. If the three devices fail independently then the state of the system in Figure 7.12 can be represented as a triple that we write as:

$$S(x, y, z) = (x, y, z).$$

Some examples of the system's Markov states are:

- S(A, A, A) means that all three devices are *alive*;
- S(A, A, F) means that devices A and B are *alive* and device C has *failed*.
- S(F, A, F) means that devices A and C have *failed* while device B is *alive*.

Next we will need the transition probabilities for the system. Again, assume that each of the devices fails independently and for simplicity that the failure process are *pure death* processes; that is, devices do not become *alive* again after failing.

Device A – has a failure probability of 0.0001.

Device B – has a failure probability of 0.0001

Device C – has a failure probability of 0.001

The state diagram is shown in Figure 7.13. There are three devices, each of which can be in two states and so this gives us $2^3 = 8$ states.

The next step is to calculate the transition probabilities. The following are examples of how to calculate a sample of the transition probabilities. Firstly the transitions from S(A, A, A).

$$\begin{aligned}
\pi_0 &= P\{A\ fails\ and\ B\ fails\ and\ C\ alive\} \\
&= 0.0001 \times 0.0001 \times (1 - 0.001) \\
&= 9.99 \times 10^{-9}
\end{aligned}$$

$$\begin{aligned}
\pi_1 &= P\{A\ alive\ and\ B\ alive\ and\ C\ fails\} \\
&= (1 - 0.0001) \times (1 - 0.0001) \times 0.001 \\
&= 9.98 \times 10^{-4}
\end{aligned}$$

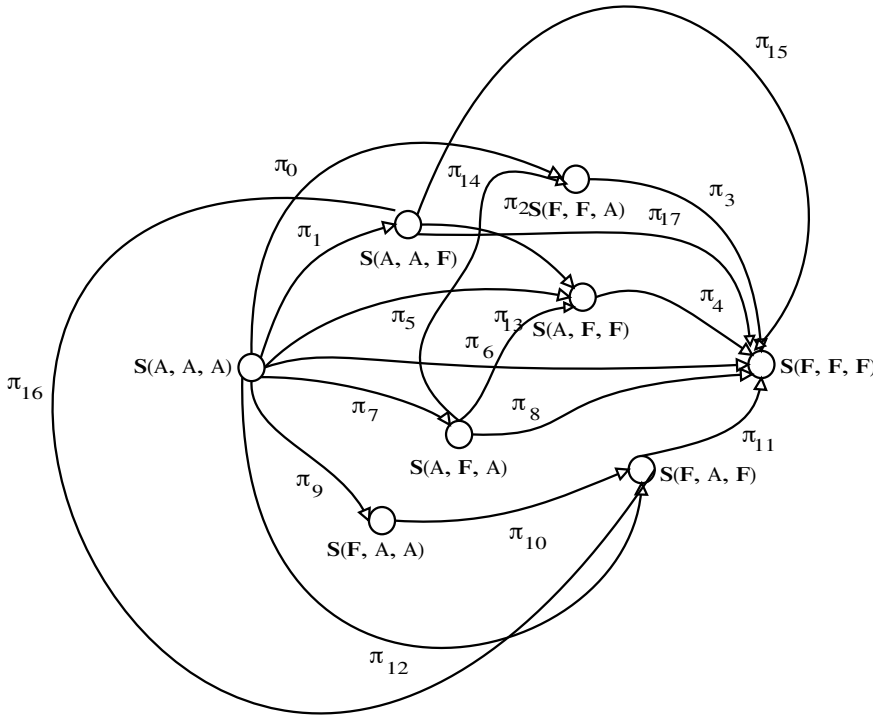
Likewise

$$\begin{aligned}
\pi_5 &= 9.999 \times 10^{-8} \\
\pi_6 &= 1 \times 10^{-11} \\
\pi_7 &= 9.989 \times 10^{-5} \\
\pi_9 &= 9.989 \times 10^{-5} \\
\pi_{12} &= 9.999 \times 10^{-8}
\end{aligned}$$

Of course we also need the transition $S(A, A, A) \rightarrow S(A, A, A)$ which is

$$\begin{aligned}
&(1 - 0.0001) \times (1 - 0.0001) \times (1 - 0.001) \\
&= 1 - \pi_0 - \pi_1 - \pi_5 - \pi_6 - \pi_7 - \pi_9 - \pi_{12} \\
&= 0.9988
\end{aligned}$$

Putting all of the transition probabilities into the transition matrix for the system gives the following matrix.



Important Note: For simplicity the diagram shows only the failure transitions. To complete the set you must add all of the transitions that leave the system in the same state.

Figure 7.13: The state transition diagram for reliability for the three device system.

$$\begin{bmatrix} 0.9988 & 0.0010 & 0.0000 & 0.0001 & 0.0000 & 0.0001 & 0.0000 & 0.0000 \\ 0.0000 & 0.9998 & 0.0001 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0001 \\ 0.0000 & 0.0000 & 0.9999 & 0.0000 & 0.0001 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0010 & 0.9998 & 0.0001 & 0.0000 & 0.0001 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 1.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.9989 & 0.9989 & 0.0001 & 0.0010 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0010 & 0.0000 & 0.9990 & 0.0000 \\ 0.0000 & 0.0001 & 0.0000 & 0.0000 & 0.0001 & 0.0000 & 0.9990 & 0.9999 \end{bmatrix}$$

A graph of the probability of being in state S(A, A, A) after 4000 time units appears in Figure 7.14.

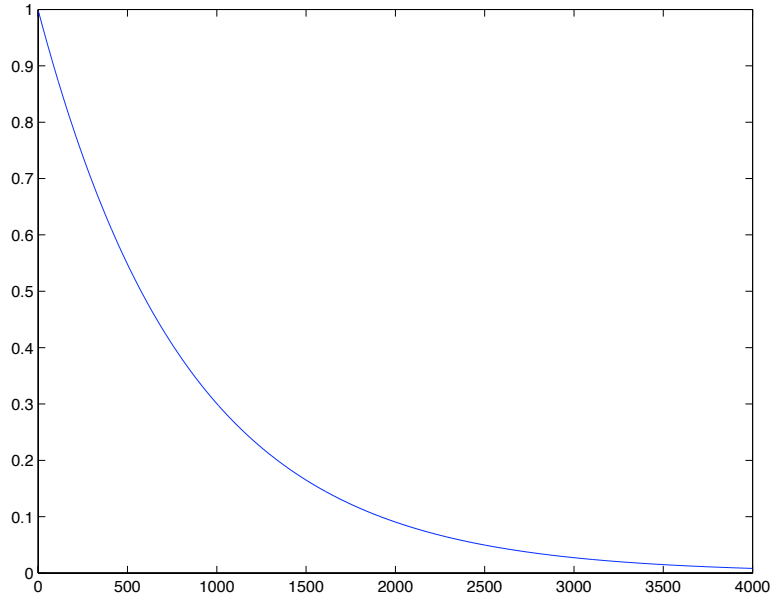


Figure 7.14: A graph of the probability of making a transition from S(AAA) to S(AAA) after 4000 time units.

7.6 Software Reliability Growth

Recall from Definition 50 that reliability is the probability of failure per unit time in a specific operational environment. Reliability growth is a collection of techniques for estimating reliability as a program is being developed and tested.

The idea comes from system reliability theory where components are tested and their *failure rates* are measured. The *failure rate* for a component is the number of failures observed per unit time. The failure rates are plotted against *reliability models* to determine how much more development time is required to reach acceptable levels of reliability.

We can summarise the steps from Table 7.1 as follows:

Measure Reliability thorough random testing. Random testing in the case of reliability growth modelling is not specifically aimed at uncovering faults in a program, although it will certainly help to uncover any faults. Rather, it is aimed at providing a random sample of inputs for the purpose of estimating a program's reliability.

Reliability Growth Models make the assumption that as development and testing continue the failure rates experienced should decrease. If the failure rates are decreasing then the reliability should be increasing.

If the failure rates are not decreasing then there may be something wrong with our testing and development process.

These notes will begin by looking at the basic reliability models and then how to estimate their parameters. A reliability model characterises software reliability, failure intensity or mean time to failure as a function of failures experienced or as a function of time. Parameter estimation becomes important in order to make full use of the predictive ability of models.

One of the big problems with reliability models for software, and certainly one that its detractors point to frequently, is the amount of error inherent in any parameter measurement. Error can make the predicative nature of reliability growth models estimates meaningless if not carefully controlled.

The Basic Execution Time Model

The reliability model that we will look at is just one of many in the field, but has been widely applied and provides a feel for what reliability modelling can achieve.

The *basic execution time* model is one of the earliest software reliability models and is due to Musa *et al.* It assumes a simple exponential relationship between failure intensity, execution time and reliability and is has been shown to be incredibly robust in practice.

Firstly, the basic execution time model assumes that reliability depends on:

- The failure intensity $\lambda(\tau)$ at time τ and
- the execution time τ itself.

Failure intensity is defined as follows.

> **Definition 60** The failure intensity λ in the basic execution time model is defined as the failures experienced per unit time.

The relationship between failure rates and reliability is given by:

$$R(\tau) = e^{-\lambda(\tau)} \tag{7.3}$$

in which e is Euler's number (the base of the natural logarithm).

Note, reliability is approximated by an inverse exponential function of time and failure intensity. Therefore:

- As the failure intensity λ approaches 0 then $R(\tau)$ approaches 1; and
- As the failure intensity approaches ∞ then $R(\tau)$ approaches 0.

This is what we would normally expect.

There is, however, another important observation to make about reliability in the basic execution time model — even if we have reached a low reliability estimate, the longer that a system is required to execute without failure, the lower the reliability $R(\tau)$ becomes.

To use the basic execution time model an estimate of the failure rate parameter λ is required. To understand the parameters and the application of the model we will need to go into some depth about concerning *measurement of failures* and its associated error.

Obtaining Failure Data

In software, the the source of most software failures are typically design faults and this is true even for failures resulting from interaction with the environment. The chief means of *fault detection* is through *testing* programs. Our testing methods in previous chapters have focused on partitioning the input domain or examining the program structure in order to find faults.

In reliability, test cases are selected randomly, however, the word random *does not mean selecting test cases completely arbitrarily*. The word random here means that we cannot predict what the next test case selected will be. In reliability measurement, to get meaningful results, test cases are selected according to the patterns of usage of the program.

The first problem is then to characterise failures in such a way that failure intensity can be estimated. In turn, estimating failure intensity requires that we record the times of failures. There are essentially four ways of doing this.

A. The time of the failure;
B. The failures experienced in a specified time interval;
C. The time interval between failures;
D. The cumulative failures experienced up to a specified time.

The difficulty with any of these time measurements is that they cannot be obtained with any certainty – executing the program on different test sets or at different times may give very different results.

Table 7.5 shows *failure time* and *time intervals between failure* data collected for a program being tested and faults removed. The assumption here is that when a program fails, that is, when the actual results deviate from the expected results, we record the time at which the failure occurred and then *remove the underlying fault*.

Thus, the second failure in Table 7.5 at $T = 19$ is independent of the first failure at $T = 10$. The assumption is not so strict and the basic execution time model has been shown to be robust in practice even if some failures do not strictly meet this assumption. Table 7.6 shows *cumulative failure time* and *failures experienced in an interval*.

The next step is to execute the program on a large number of test inputs, chosen at random according the operational profile, in order to estimate the *failure probability distribution* for the program. The failure data and probability distributions are shown in Table 7.7.

Operational Profiles

Operational profiles are important for reliability growth testing. They relate programs back to the environment in which programs execute. For reliability growth, we will need to look at operational profiles in more depth.

Although many programs execute in finite time finite time may still mean much longer time spans than the few seconds it takes to execute a typical desktop application. Programs may be required to execute for weeks or months, or even years in the case of embedded systems.

Failure number	Failure time (sec.)	Failure interval (sec.)
1	10	10
2	19	9
3	32	13
4	43	11
5	58	15
6	70	12
7	88	18
8	103	15
9	125	22
10	150	25
11	169	19
12	199	30
13	231	32
14	256	25
15	296	40

Table 7.5: Failure time and failure interval data.

Time (sec.)	Cumulative Failures	Failures in Interval
30	2	2
60	5	3
90	7	2
120	8	1
150	10	2
180	11	1
210	12	1
240	13	1
270	14	1

Table 7.6: Cumulative failures and cumulative failures in an interval.

Failures in time period.	Elapsed Time $t_A = 1\text{hr}$	Elapsed Time $t_B = 10\text{hr}$
0	0.10	0.01
1	0.18	0.02
2	0.22	0.03
3	0.16	0.04
4	0.11	0.05
5	0.08	0.07
6	0.05	0.09
7	0.04	0.12
8	0.03	0.16
9	0.02	0.13
10	0.01	0.10
11	0	0.07
12	0	0.05
13	0	0.03
14	0	0.02
15	0	0.01

Table 7.7: The failure probability distribution for the data in Table 7.5.

Typically many different functions are implemented in a single system. For example, an editor has functions for inserting text, inserting graphics, deleting text and moving text around. In order to arrive at an operational profile for a program, or system, the functions can be performed by the system need to be identified because we will view each execution of a program as choosing and executing some subset of these functions.

Each function has associated with it an input domain much like in the case of our previous experience with testing. The big difference is that in the case of operational profiles we want to determine the set of inputs that will cause a function of the program to be executed. We now have the following definitions.

Definition 61

(i) An *input state* is an element of the input domain that triggers the execution of one of the program's functions.

(ii) The *operational profile* of the program is the set of input states for the program and the probability with which each input state can occur in the program environment.

Determining the operational profile can be done via manual estimation, by running the program a number of times to get an approximation of the amount of times different inputs states occur, or by looking at the characterisation of the input. For example, consider a word processor. It is likely that the input of entering a single character will occur with a higher probability than the input of inserting an image. We can estimate the operational profile by looking at documents and counting the number of characters and number of images in these documents.

Time

Reliability is measured with respect to *time*. There are three kinds of time that is involved in reliability engineering.

(i) **Execution Time** – the time spent by the processor in executing the instructions of the program;

(ii) **Calendar Time** – the normal passage of time measured in hours, days, and months;

(iii) **Clock Time** – the time elapsed from the start of program execution and includes the wait and execution time of other programs.

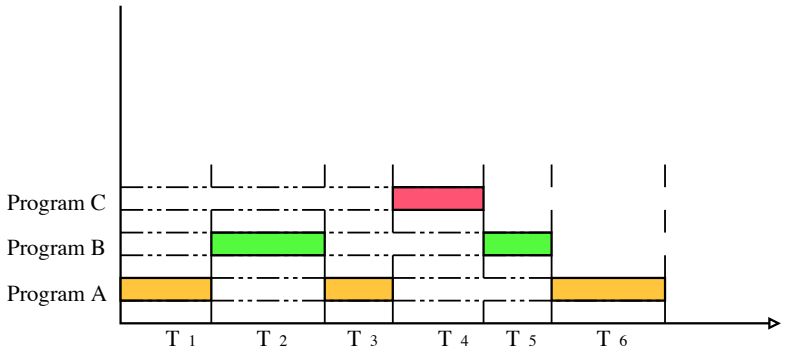


Figure 7.15: Measuring the execution time of program A.

Consider the program in Figure 7.15. The *execution time* for program A is $T_1 + T_3 + T_6$ while the *elapsed time* is $T_1 + T_2 + T_3 + T_4 + T_5 + T_6$. The problem is in actually determining the execution time for a program, precisely because in any operating system that schedules processes in a fair manner, other programs will use the processor during the execution of the program.

The result is that timing measurements are rarely repeatable and often show up as a distribution of times normally distributed around a mean. This is one chief source of error in estimating model parameters. While the model has some tolerance for error in measuring the time, choosing the right clock to record failure times is still important.

Secondly, there is an issue of *scale*. Errors due to timing measurements have significantly less impact if failure times are measured at small time scales and failures occur at large time scales. For example, an error in time

measurement of about 100 milli-seconds hardly influences the basic execution time model if the time interval between failures is in the order of days.

The time measured in hours, days and months is more usual for project planning. Although we will not look at how to relate time measurements in terms of clock time or calendar time back to execution time, it is possible to make execution time estimates of these time values.

Understanding Reliability Growth

Now that we have looked again at failures, operational profiles and the problem of measuring time we will return to the general idea of reliability growth. The idea is summed up in the graph in Figure 7.16.

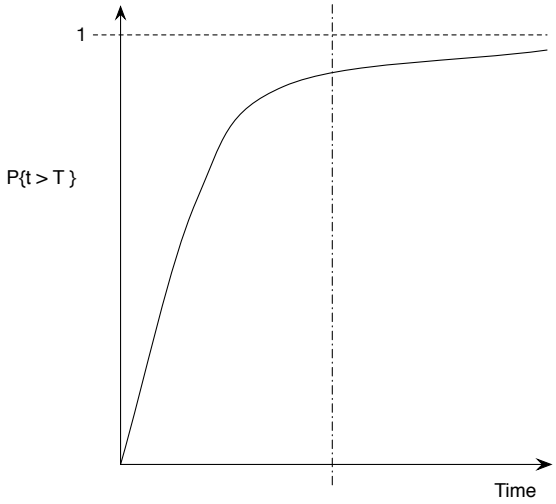


Figure 7.16: Reliability growth as a function of time.

What we observe is that at the start of the testing phase our reliability starts low. Remember that reliability is the probability of failure free operation for a given time in a given environment. Now, the idea of operating without failure for a given time T can be expressed by saying that the first failure that we encounter occurs after time T, or in other words we are seeking:

$$P\{\tau > T\} \tag{7.4}$$

where τ is the time at which we observe the first failure.

As testing continues, more faults are removed from the program and so the probability $P\{\tau > T\}$ increases. Note, that we are not changing T here; the random variable is the time at which the first failure is observed. Of course, we should also observe a drop in failure intensity as reliability increases as in Figure 7.17.

Putting the reliability and failure intensity graphs together yields the graph in Figure 7.18.

Back to the Basic Execution Time Model

There are a number of assumptions in the basic execution time model.

A. The model is based on execution time.

B. When failures are detected it is assumed that the underlying faults are removed and execution continued. The model assumes that the achieved rate of failure reduction is *constant*.

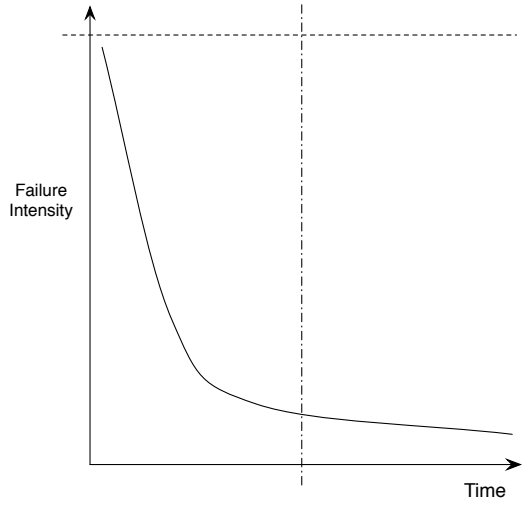


Figure 7.17: The failure intensity as a function of time when reliability is increasing.

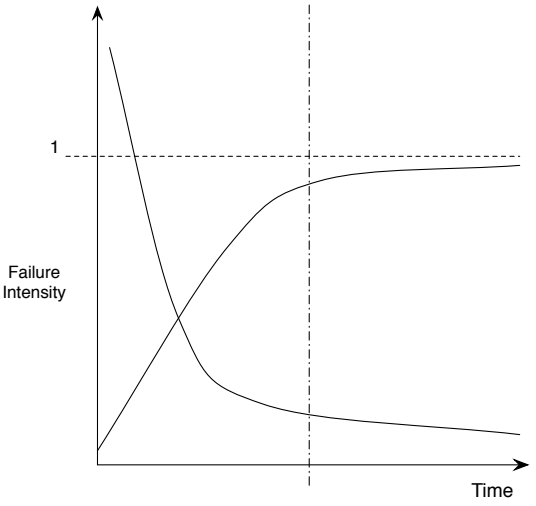


Figure 7.18: Reliability and Failure intensity superimposed.

C. The operational profile is uniform – every input occurs with equal probability. Despite this, the model has proved to be robust in practice when used with operational profiles that do not fit this assumption.

An immediate consequence of the assumption (B) is that failure intensity falls off linearly with the number of failures observed. In fact, the assumption gives a graph like that in Figure 7.19.

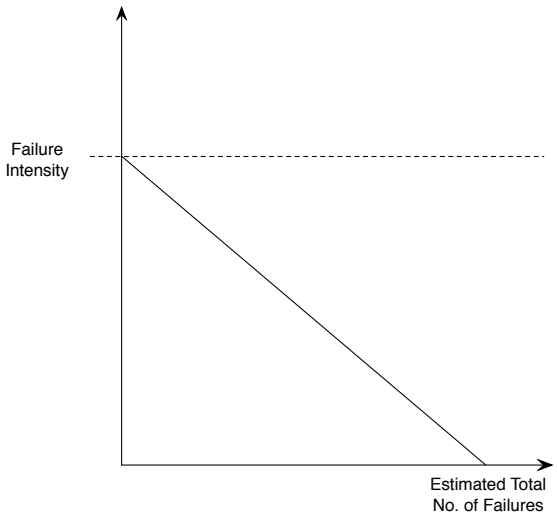


Figure 7.19: Assumption (B) - The rate of failure reduction is constant.

From the graph, *failure intensity* λ is a function of the failures experienced, or observed, μ and is given by a linear relationship:

$$\lambda(\mu) = \lambda_0 \left(1 - \frac{\mu}{\nu_0}\right) \tag{7.5}$$

where λ_0 is the initial failure intensity at the start of the execution period and the quantity ν_0 is the total number of failures that would occur in infinite time.

Both λ_0 and ν_0 are parameters to the model. The task then becomes one of estimating λ_0 and ν_0 from the actual failure data obtained from measuring the program.

Example 62

> If we assume that we have estimated an initial failure intensity (failures/unit time) of 10 failures/hr (λ_0), and that our estimated total number of failures is 100 failures (ν_0) and we have experienced 50 failures:
>
> $$\lambda(50) = 10(1 - \frac{50}{100}) = 5 \text{ failures/hr}.$$
>
> Suppose that after some further fault repair and testing we have now experienced a total of 70 failures. Then our failure intensity is predicted to be 3 failures/hr.

The basic model also allows us to estimate the *total failures as a function of execution time*:

$$\mu(\tau) = \nu_0 \left(1 - e^{\left(-\frac{\lambda_0}{\nu_0}\tau\right)}\right) \tag{7.6}$$

which yields an expression for the mean (or expected) number of failures experienced up to time τ.

It allows us to estimate the *failure intensity as a function of execution time*.

$$\lambda(\tau) = \lambda_0 e^{\left(-\frac{\lambda_0}{\nu_0}\tau\right)} \tag{7.7}$$

and it allows us to estimate *reliability as a function of time*.

$$R(\tau) = e^{-\lambda(\tau)} \tag{7.8}$$

Example 63

Consider again a program with initial failure intensity of 10 failures/hr and an estimated 100 failures in total.

The total failures is after 10 CPU hours is:

$$\begin{aligned}
\mu(10) &= \nu_0 \left(1 - e^{\left(-\frac{\lambda_0}{\nu_0}\tau\right)}\right) \\
&= 100 \left(1 - e^{\left(-\frac{10}{100}10\right)}\right) \\
&= 100 \left(1 - e^{(-1)}\right) \\
&= 100(1 - 0.368) \\
&\approx 63 \text{ failures}
\end{aligned}$$

After 100 CPU hours it is:

$$\begin{aligned}
\mu(100) &= 100 \left(1 - e^{\left(-\frac{10}{100}100\right)}\right) \\
&= 100 \left(1 - e^{(-10)}\right) \\
&= 100(1 - 0.0000454) \\
&\approx 100 \text{ failures} \qquad \text{(well almost!)}
\end{aligned}$$

Choosing Failure Objectives

Once we have this basic machinery in place the model can be used for *evaluating programs* and *scheduling testing*. At any time in the project we can choose a *failure objective*; although this is better done earlier than in testing. The failure objective can be: a specified reliability estimate for the program; or a specified failure intensity for the program. We can choose reliability objectives for specific builds of a program or we can choose reliability objectives for the entire program. The basic execution time model can then be used to predict how much many more failures we need to observe and repair, or how much more execution time is required to reach the target reliability (see Figure 7.20).

To see how this works, suppose that we choose λ_0 and ν_0 to fit with our current observed failure intensities. Also assume that we have chosen a failure intensity objective for our project. To achieve a reliability objective we will need to change the failure intensity from our present intensity λ_P to our desired failure intensity λ_F.

It is straight-forward to calculate λ_F from the failures already experienced by using the linear relationship in Equation 7.5 to yield:

$$\Delta\mu = \frac{\nu_0}{\lambda_0}(\lambda_P - \lambda_F) \tag{7.9}$$

So, once we have experienced the additional failures $\Delta\mu$ our model predicts that the desired failure intensity has been reached.

How much more time does the program need to be executed – and to detect failures and remove faults – before the target failure intensity is reached? Again, the answer comes from the relationship in Equation 7.5.

$$\Delta\tau = \frac{\nu_0}{\lambda_0} \ln \frac{\lambda_P}{\lambda_F} \tag{7.10}$$

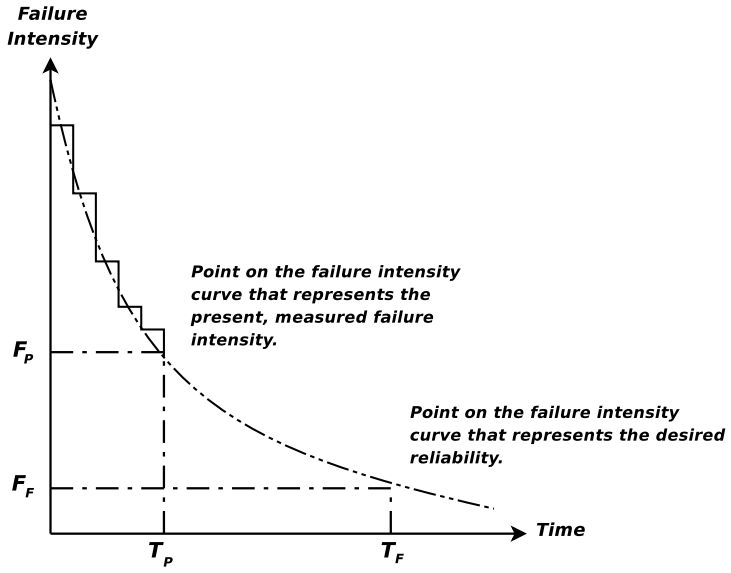


Figure 7.20: Predicting the amount of development time and testing based on reliability models.

Estimating the Parameters for the Model

Before being able to apply the model however, we still need to either *predict* or *estimate* the values of the model parameters : the total number of failures ν_0, and the initial failure intensity λ_0. We will focus on estimation rather than predication.

You can estimate the parameters of the model in a number of ways. The simplest is to carry out the following steps.

Step 1 Begin by executing the program and recording each failure, its execution time and any other information relevant to the failure.

Step 2 Plot the failure intensities (failures per unit time) against the number of failures experienced; *or* plot failure intensity versus execution time;

Step 3 The two parameters required to estimate the equation of the line in Figure 7.19 are λ_0 and ν_0 and these parameters can be chosen to maximise the likelihood of the observed set of failure intensities.

Now there are a number of choices for choosing the parameters to parameters ν_0 and λ_0 to maximise the likelihood of observing the failure intensities.

(1) Maximum likelihood estimators (described in Appendix B);

(2) Least squares estimators; and

(3) Error seeding models (a forerunner of mutation testing).

Least Squares Estimation

We can use the failure intensity versus number of failures experienced to estimate λ_0 and ν_0 by choosing the line of best fit through the data. Note that a line is not always optimal and may give us inaccurate results.

Note: the equation of the line of best fit comes directly from Equation 7.5:

$$\lambda(\mu) = \lambda_0 - \frac{\lambda_0}{\nu_0}\mu.$$

The aim in least squares estimation is to minimise the distance between each data point and the line of best fit passing through all of the data points.

The formulae for calculating least squares estimates for the parameters of the basic model are as follows. We will use μ_i for the number of failures experienced, and λ_i for the failure intensities.

$$\lambda_0 = \overline{\lambda} - \frac{\lambda_0}{\nu_0}\overline{\mu} \tag{7.11}$$

$$\frac{\lambda_0}{\nu_0} = \frac{\Sigma(\mu_i - \overline{\mu})(\lambda_i - \overline{\lambda})}{\Sigma(\mu_i - \overline{\mu})^2} \tag{7.12}$$

where $\overline{\lambda} = \frac{1}{n}\Sigma\lambda_i$ is the mean of the observed failure intensities and $\overline{\mu} = \frac{1}{k}\Sigma\mu_i$. An estimate on the error for each of the estimated λ_i is given by

$$S^2 = \frac{1}{n-2}[\Sigma(\lambda_i - \overline{\lambda})^2 - \frac{\lambda}{\nu_0}^2 \Sigma(\mu_i - \overline{\mu})^2] \tag{7.13}$$

Numerous calculators are available on the web for computing least squares estimates, errors, and linear regression models.

Of course we need to note some of the sources of error in our predictions. Some sources of error are listed below.

- **Unobserved Failures:** Failures may be missed or not correctly reported. In general this means that failure intensity is going to be underestimated.
- **Observing Time Measurements:** Observing execution time or measuring execution time typically results in uncertainty. The very act of measuring the execution of a program changes it.
- **Evolving Programs:** The reliability models assume that programs do not change except for the failure detection and repair process. However, in practice systems are adapted and change over time; reliability measurements taken at the test and release phases do not necessarily carry over to maintenance phases.

In their book (*Software Reliability: Measurement, Prediction, Application*), Musa *et al.* argue that the linear models obtained by least squares approximations are biased by large values of λ, but are adequate when the sample size is relatively small. We will use linear estimation in these notes. If you are interested in alternatives then look at *maximum likelihood estimation*, which is outlined in Appendix B.

Calculating the Model Parameters – An Example

Consider the data in Table 7.8.

Musa *et al.* give a method for determining failure intensity from tabulated failure data.

Step 1 Let the total time for observing the program be T. First choose a k and record the times of every k^{th} failure, that is, the times at which we have k failures, $2k$ failures, $3k$ failures and so on, until the end of the time period T. It is common to have less than k failures in the final interval.

Step 2 Partition T in p disjoint time intervals $(\tau_{(n-1)k}, \tau_{nk}]$ where each $\tau_n k$ corresponds to the time at which nk failures occurred.

Step 3 The observed failure intensity for the n^{th} interval is simply

$$r_n = \frac{k}{\tau_{nk} - \tau_{(n-1)k}}$$

if n is not the last interval and

$$r_n = \frac{T - k(p-1)}{T - \tau_{(p-1)k}}$$

in the final interval.

Failure number	Failure time (sec.)	Failure interval (sec.)
1	10	10
2	19	9
3	32	13
4	43	11
5	58	15
6	70	12
7	88	18
8	103	15
9	125	22
10	150	25
11	169	19
12	199	30
13	231	32
14	256	25
15	296	40

Table 7.8: Time based failure data.

For the data in Table 7.8 above we will choose $k = 3$ and so we have 5 intervals as in Table 7.9. Table 7.10 now

Failure number	Failure time (sec.)	Failure interval (sec.)	λ
1	10		
2	19		
3	32	32	0.094
4	43		
5	58		
6	70	38	0.079
7	88		
8	103		
9	125	55	0.055
10	150		
11	169		
12	199	74	0.041
13	231		
14	256		
15	296	97	0.031

Table 7.9: Failure time and failure interval data.

shows the failure intensities and the number of failures experienced. The regression gives us the equation

$$\lambda(\mu) = 0.19 - 0.00055\mu$$

from which we determine that $\lambda_0 = 0.19$ and $\nu_0 = 345.45$ which we round to 345.

For the data given in Table 7.10 we now get the following relationships between failure intensity and failures experienced (7.14),

$$\lambda(\mu) = 0.19 - 0.00055\mu \tag{7.14}$$

failure intensity and execution time (7.15),

$$\lambda(\tau) = 0.19e^{(-0.00055\tau)} \tag{7.15}$$

Failures Experienced	Failure Intensity (Failures/sec)
3	0.094
6	0.079
9	0.055
12	0.041
15	0.031

Table 7.10: Failure times and the number of failures experienced.

and mean number of failures and execution time (7.16)

$$\mu(\tau) = 345\left(1 - e^{[-0.00055\tau]}\right). \tag{7.16}$$

What is the estimated execution time required to achieve a failure rate of 0.015 *failures/sec*?

In this case we have $\lambda_P = 0.031$ failures/sec and we wish to achieve $\lambda_F = 0.015$ failures/sec. Using the model above we can calculate:

$$\begin{aligned} \Delta\tau &= \frac{\nu_0}{\lambda_0} \ln \frac{\lambda_P}{\lambda_F} \\ &= 1818.18 \ln 2066.67 \\ &= 573.21 \quad \text{seconds.} \end{aligned}$$

In that time we would need to experience a further

$$\begin{aligned} \Delta\mu &= \frac{\nu_0}{\lambda_0}(\lambda_P - \lambda_F) \\ &= 1818.18 * 0.016 \\ &= 29.09 \quad \text{or} \quad 29 \text{ failures.} \end{aligned}$$

Some Final Words on Reliability Testing

In reliability we are seeking test inputs that will give us the most rapid reduction in failure intensity with respect to execution time. The idea here is one of *efficient testing* which is related to testing costs. If the operational profile for the program matches the program's actual usage then the failures that would have been most frequent in practice are *detected* and removed first. Thus, we gain our efficiency in testing.

Test inputs selected according to the operational profile often require a *testing oracle* to determine if the test results in a failure or not.

7.7 Error Seeding

Another method for evaluating reliability is the *error seeding* approach. Error seeding is actually a forerunner of mutation testing strategies. It is an approach for *estimating* the total number of failures remaining in a program given a small *seeded* sample of errors.

Its most practical application (due to Harland Mills of IBM in the late '70s and early '80s) is to estimate the number of faults remaining in a program under test at the end of each test phase. Reliability increases if the number of estimated remaining faults decreases.

Let us assume that there are N faults in the program. The aim of error seeding is to estimate N by seeding the program with E faults and then using ratio of the number of seeded faults detected to estimate N. The method for error seeding works as follows.

Step 1 Randomly seed the program with a number E of *known* faults.

Step 2 Test or inspect the program logging the faults that are uncovered and noting the time.

Step 3 Suppose that, of the faults detected, we can determine that P of these were seeded faults and that Q were unseeded faults. Then the maximum likelihood estimate of N is:

$$\begin{aligned} \frac{P}{E} &= \frac{Q}{N} \\ N &= \frac{E \times Q}{P} \end{aligned} \tag{7.17}$$

Error seeding assumes that the seeded faults and the indigenous faults are equally likely to occur. This is the problem in the error seeding model because the implication is that seeded faults are somehow representative of unseeded faults, which is not always the case. However, no assumptions are made about the initial failure intensities or similar model parameters so estimates are independent of the basic model.

How confident are we in the results of the prediction?

The *confidence* that we have in our estimate of N, the number of program faults, is the probability that we will reject an incorrect value for N using the estimate.

Given our estimate N let us assert that the program has no more than K. The formula for calculating confidence in this assertion is:

$$C = \begin{cases} 1 & \text{if } N > K \\ \left(\begin{array}{c} E \\ P-1 \end{array} \right) / \left(\begin{array}{c} E+K+1 \\ K+P \end{array} \right) & \text{if } N \leq K \end{cases} \tag{7.18}$$

where

$$\left(\begin{array}{c} A \\ B \end{array} \right) = A!/(B! \times (A-B)!)$$

An Error Seeding Example

Consider the testing of a small hypothetical program and assume that, at a certain point in testing, we have deliberately seeded 10 faults. The next round of testing without repair uncovers 6 of the deliberately seeded faults and 4 indigenous, or unseeded, faults.

According to the error seeding model an estimate of the total number of faults in the program is

$$N = \frac{10 \times 4}{6} = 6\frac{2}{3} \approx 7$$

What confidence do we have in this estimate? Let us assert that the program has no more than $K = 7$ faults. Calculate C by:

$$\begin{aligned} C &= \frac{10!}{5! \times 5!} \Big/ \frac{18}{13! \times 5!} \\ &= 0.03 \end{aligned}$$

or about a 3% probability that there no more than 7 faults in total in our program. Alternatively, we can say that we have 97% confidence that there are no more than 7 faults in the program.

7.8 References

[1] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, 1987.

[2] *The Handbook of Software Reliability Engineering*, IEEE Computer Society, McGraw-Hill, 1996.

[3] C. M. Krishna and K. G. Shin, *Real-Time Systems*, McGraw-Hill, 1997.

[4] R. Denney, *Succeeding with use cases: working smart to deliver quality*, Pearson Education India, 2005.

Chapter 8

Security Testing

In this chapter, we will look at a few systematic and automated approaches to testing security of systems. Specifically, we will look at three different ways of *penetration testing*.

Penetration testing (or *pentesting*) is the process of attacking a piece of software with the purpose of finding security vulnerabilities. In practice, pentesting is just hacking, with the difference being that in pentesting, you have the permission of the system owners to attack the system.

In other words, penetration testing is *hacking with permission*.

One way to perform penetration testing is to just try out a whole load of tricks that have worked before. For experienced pentesters, this is an excellent and fruitful approach; and in fact, many people make their career out of intelligently trying to break systems. However, this is exceedingly difficult to teach in a few week period off a university subject. Instead, in this chapter, we are going to focus on *automated* pentesting, which is generally called *fuzzing* or *fuzz testing*, which is a systematic and repeatable approach to pentesting, and one which is used by many great penetration testers.

8.1 Introduction to Penetration Testing

First, some terminology. The aim of penetration testing is to find *vulnerabilities*. A vulnerability is a security hole in the hardware or software (including operating system) of a system that provides the possibility to attack that system; e.g., gaining unauthorised access. Vulnerabilities can be weak passwords that are easy to guess, buffer overflows that permit access to memory outside of the running process, or unescaped SQL commands that provide unauthorised access to data in a database.

Example 64 *As an example of an SQL vulnerability, consider a request that allows a user to lookup their information based on their email address via a textbox, resulting in the following query to a database:*

```
SELECT *
FROM really_personal_details
WHERE email = '$EMAIL_ADDRESS'
```

in which `$EMAIL_ADDRESS` *is the text copied from the text box.*

If the text `myemail@somedomain.com` *is entered, then this works fine. However, if the user is aware of such a vulnerability, they may instead enter:* `myemail@somedomain.com' OR '1=1` *(note the unopened and unclosed quotation marks here). If the developers have not been careful, this* may *result in the following query being issued:*

```
SELECT *
```

```
FROM really_personal_details
WHERE email = 'myemail@somedomain.com' OR '1=1'
```

The `WHERE` *clause evaluates to* $true$*, and therefore, the user will be able to gain unauthorised access to all data in the table, including that of other users.*

Buffer overflows are another common security vulnerability. These occur when data is written into a buffer (in memory) that is too small to handle the size of the data. In some languages, such as C and C++, the additional data simple overwrites the memory that is located immediately after the buffer. If carefully planned, attacker-generated data and code can be written here.

Example 65 *Consider the following example of a stack buffer overflow*[1].

The following C program takes a string as input, and allocates it to a buffer 12 characters long:

```
void foo (char *bar)
{
  char  c[12];

  strcpy(c, bar);  // no bounds checking
}

int main (int argc, char **argv)
{
  foo(argv[1]);
}
```

Now, consider the two cases in Figure 8.1[2]*. On the left is the case where the input string,* `hello\0`*, is small enough to fit into the buffer. The frame pointer and return address for the* `foo` *function sit next to the buffer in memory. On the right side is an example where the input string,* `AA...\x08\x35\xC0\x80`*, is longer than the buffer. This writes over the frame pointer and the return address. Now, when* `foo` *finishes executing, the execution will jump the address specified in the return address spot, which is at the start of* `char c[12]`*. In this case, the contents of* `char c[12]` *is meaningless, but in an attack, the string would contain payload that could be a malicious program, which would have privileges equivalent to the program being executed; e.g., it could have superuser/admin access.*

Clearly, there are many ways to try to reduce the chance such attacks: defensive programming that checks for array bounds, using programming languages that do this automatically, or finding and eliminating these problems. We can find these problems using reviews and inspections, letting hackers find them for us (not recommended), or using verification techniques.

In this chapter, we look at verification techniques, specifically, fuzz testing.

> **Definition 66** *Fuzz testing* (or *fuzzing*) is a (semi-)automated approach for penetration testing that involves the randomisation of input data to locate vulnerabilities.

Typically, a fuzz testing tool (or *fuzzer*) generates many test inputs and monitors the program behaviour on these inputs, looking for things such as exceptions, segmentation faults, and memory leaks; rather than testing for functional correctness. Typically, this is done live: that is, one input is generated, executed, and monitored, then the next input, and so on.

We will look at three techniques for fuzzing:

1. Random testing: tests generated randomly from a specified distribution.
2. Mutation-based fuzzing: starting with a well-formed input and randomly modifying (mutating) parts of that input

[1] Taken from `https://en.wikipedia.org/wiki/Stack_buffer_overflow`
[2] Taken from `https://en.wikipedia.org/wiki/Stack_buffer_overflow`

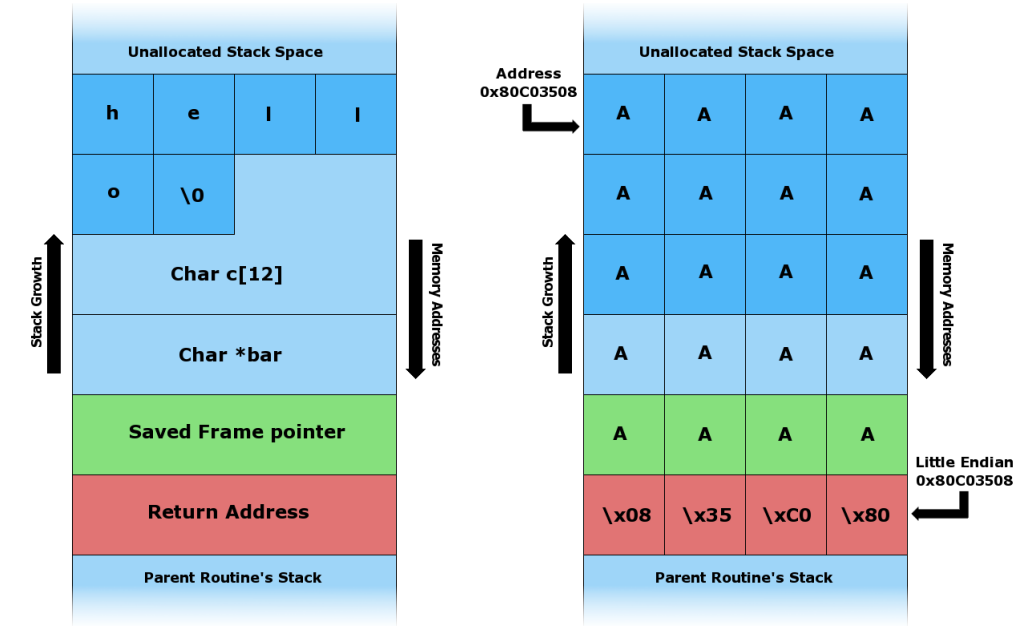


Figure 8.1: An example of expect buffer usage (left) and overflow (right)

3. Generation-based fuzzing: using some specification of the input data, such as a grammar of the input.

8.2 Random Testing

Random testing is one approach to fuzzing. We have already discussed random testing in Chapter 7 (Section 7.3. As with random testing for reliability, tests are chosen according to some probability distribution (possible uniform) to permit a large amount of inputs to be generated in an fast and unbiased way.

Advantages

- It is often (but not always) cheap and easy to generate random tests, and cheap to run many such tests automatically.
- Is is *unbiased*, unlike tests selected by humans. This is useful for pentesting because the cases that are missed during programming are often due to lack of human understanding, and random testing may search these out.

Disadvantages

- A prohibitively large number of test inputs may need to be generated in order to be confident that the input domain has been adequately covered.
- The distribution of random inputs simply misses the program faults (recall the Pareto-Zipf principle from Chapter 1).
- It is highly unlikely to achieve good coverage.

The final point deserves some discussion. If we take any non-trivial program and blast it with millions of random tests, it is unlikely that we will achieve good coverage, where coverage could be based on any reasonable criteria (control-flow, mutation, etc.).

Consider the following example, in which we have a "fault" at line 7 where the program unexpectedly aborts.

```
1. void good_bad(char s[4]) {
2.   int count = 0;
3.   if (s[0] == 'b') count++;
4.   if (s[1] == 'a') count++;
5.   if (s[2] == 'd') count++;
6.   if (s[3] == '!') count++;
7.   if (count >= 3) abort();  //fault
8. }
```

This program will only fail on the input "bad!". Using random testing, random arrays of four characters will be generated, but encountering the fault at line 7 is highly unlikely. If we consider a 32-bit architecture, the probability of randomly generating the string "bad!" requires us to randomly choose 'b' at index 0 (probability of 1 in 2^{32}, 'a' at location 1 (probability of 1 in 2^{32}), etc., leaving us with the probability of 1 in 2^{128} that the code will be executed. In fact, almost every randomly generated test will execute false for every branch.

While this example is a fabricated example, such cases are common place in real programs. For example, consider any branch of the form `x == y` (unlikely that `x` and `y` will have the same value). Or, more interestingly, consider the chance of randomly generating a correct username and password to get into a system, or generating the correct checksum for a packet of data that is received. This latter case is common in security-critical applications: only if we have the correct checksum can be "do something interesting":

```
 1. boolean my_program(char [] data, int checksum)
 2  {
 3.   if (calc_checksum(data) == checksum) {
 4.     //do something interesting
 5.     return true;
 6.   }
 7.   else {
 8.     return false;
 9.   }
10. }
```

In cases such as this, a random testing tool will likely spend most of its time just testing the false case over and over, which is not particularly useful.

A standard way to address this is to (if possible) measure the code coverage achieved by your tests after a certain amount of time, and look for cases like these. Then, modify your random testing tool to first send some data with a correct checksum, and then use random testing for the remainder of inputs. Rinse and repeat.

8.3 Mutation-based Fuzzing

Mutation-based fuzzing is a simple process that takes valid test inputs, and mutates small parts of the input, generating (possibly invalid) test inputs. The mutation (not to be confused with mutation analysis discussed in Section 4.3) can be either random or based on some heuristics.

As an example, consider a mutation-based fuzzer for testing web servers[3]. A standard HTTP GET request for the index page could be a valid input:

```
GET /index.html HTTP/1.1
```

[3]Taken from `http://pages.cs.wisc.edu/~rist/642-fall-2012/toorcon.pdf`

From this, many anomalous test inputs, which should all be handled by the server, can be generated by randomly changing parts of that input:

```
AAAAAA...AAAA /index.html HTTP/1.1
GET ///////index.html HTTP/1.1
GET %n%n%n%n%n%n.html HTTP/1.1
GET /AAAAAAAAAAAAA.html HTTP/1.1
GET /index.html HTTTTTTTTTTTTTP/1.1
GET /index.html HTTP/1.1.1.1.1.1.1.1
```

One can imagine a simple program being set up to generate such inputs and deliver them to a web server.

As an example of a more heuristic-based mutation fuzzer, consider a text field in a web form in which a user enters their email address, as in Example 64. Attacks such as those presented are common, so an heuristic mutation fuzzer will add targets such as that to fields, instead of random data. So, a valid input such as:

```
'myemail@somedomain.com'
```

could be mutated to things such as:

```
'myemail@somedomain.com' OR '1=1'
'myemail@somedomain.com' AND email IS NULL
'myemail@somedomain.com' AND username IS NULL
'myemail@somedomain.com' AND userID IS NULL
```

These last three attempt to guess the name of the field for the email address. If any of these give a valid response, we know that we guess the name of the field correctly; otherwise a server error will be thrown.

Advantages

The main advantages of fuzzing are compared to random testing are:

- It generally achieves higher code coverage than random testing. While issues such as the checksum issue discussed earlier still occur, they often occur less of the time if the valid inputs that are mutated have the correct values to get passed these tricky branches. Even though the mutated tests may change these, some will change different parts of the input, and the e.g., checksum will still be valid.

Disadvantages

- The success is highly dependent on the valid inputs that are mutated.
- It still suffers from low code coverage due to unlikely cases (but not to the extent of random testing).

Tool support

There are many tools to support mutation-based fuzzing.

- zzuf (`http://caca.zoy.org/wiki/zzuf`) is a commonly-used tool for "corrupting" valid input data to produce new anomaly tests. It uses randomisation, and has controllable properties such as how much of the input should be changed for each test.
- Peach (`http://www.peachfuzzer.com/`) is a well-known fuzzer that supports mutation fuzzing, and has reached a level of maturity that make it applicable to many projects.

8.4 Generation-based Fuzzing

Generation-based fuzzers (or *intelligent fuzzers*) typically generate their own input from such existing models, rather than mutating existing input (although many tools combine the two approaches).

Typically, generation-based fuzzers have some information about the format of the input that is required; for example, a grammar of the input language (e.g., SQL grammar), knowledge of the file format. Using this knowledge, a generated-based fuzzer can create inputs that preserve the structure of the input, but it can randomly or heuristically modify parts of the input based on that knowledge. Therefore, instead of randomly modifying parts of a string representing an SQL query, it can produce syntactically-correct SQL, but with random data within that structure.

Further to this, by knowing the input protocol and the interactions that are required, a generation-based fuzzer can more intelligently select inputs. For example, by knowing a protocol for a web server, it can behave as a true web client would, allowing generation of correct, dynamic responses to server responses, rather than just random data.

In essence, the strategies applied, such as which parts to change and how fine-grained the changes are, affect the "intelligence" of the fuzzer.

Example 67 *As an example, consider generating a HTTP POST call with a SOAP action:*

```
POST / HTTP/1.1
Host: unimelb.edu.au
Content-Type: text/xml; charset=utf-8
Content-Length: 372
SOAPAction: "http://wheverer.com/somewhere"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"...
<soap:Body>
  <firstName>A name</firstName>
  <lastName>Another name</lastName>
  <paramsXML>param 1</paramsXML>
</soap:Body>
</soap:Envelope>
```

A mutation-based fuzzer will take an input such as this and randomly modify parts of it; for example, by replacing some random characters. In most cases, this would result in a query that cannot be parsed by the server. Non-parsable inputs can be valuable for testing the parser, but we want to also test other parts.

A generation-based fuzzer, on the other hand, would know the HTTP and SOAP protocols, so would first generate the skeleton for a parsable query, and then generate the values in between the tags of the SOAP query (randomly or heuristically). For example, it may randomly change the value in the `firstName` tag to a much longer string. This would result in a SOAP query that is longer than specified in the `Content-Length` HTTP parameter (372 chars), perhaps resulting in a case where the SOAP query overflows the buffer that is allocated — presumably 373 chars (although one would hope that the server is using a library that does not do this!).

Advantages

- Knowledge the input protocol means that valid sequences of inputs can be generated that explore parts of the program, thus generally giving higher coverage.

Disadvantages

- Compared to random testing and mutation fuzzing, it requires some knowledge about the input protocol.

- The setup time is generally much higher, due to the requirement of knowing the input protocol; although in some cases, the grammar may already be known (e.g., XML, RFC).

Tool support

- Peach(`http://www.peachfuzzer.com/`), which is mentioned also at a mutation fuzzer, supports both mutation and generation-based fuzzing. For smarter generation-based fuzzing, it also monitors feedback sent to it via network protocols (for fuzzing e.g., web servers) to intelligent select new tests.

8.5 Memory Debuggers

A *memory debugger* is a tool for finding memory leaks and buffer overflows. Memory debuggers are important in fuzzing, especially in languages with little support for memory management.

The reason why memory debuggers are important is because anomalies such as buffer overflows are difficult to observe using system behaviour. For example, consider the stack buffer overflow presented earlier in this chapter. If the overflow is just by a few characters, it would be difficult to detect unless that particular part of memory is accessed again, which may not be the case.

Generally, memory debuggers monitor looking for four issues:

1. Uninitialised memory: references made to memory blocks that are uninitialised at the time of reference.
2. Freed memory: reads and writes to/from memory blocks that have been freed.
3. Memory overflows: writes to memory blocks past the end of the block being written to.
4. Memory leaks: memory that is allocated but no longer able to be referenced.

Memory debuggers typically work by modifying the source code at compile type to include specific code that checks for these issues; for example, by keeping track of a buffer size, and then inserting code directly before a write to check whether the memory being written is larger than the target buffer.

Performance Cost

While monitoring for these properties is useful, it is generally at a high performance cost. The overhead of keeping track of the allocated memory, plus the checks made, is expensive. For example, programs running using the well-known *Valgrind* tool run between 20-30 times slower than with no memory debugging.

Tool Support

- *Valgrind*[4] (and its tool *memcheck*) is a well-known open-source memory debugger for Linux, Mac OS, and Android. It works by inserting monitoring code directly into the source, and replaces the standard C memory allocation tools (e.g., alloc, malloc) with its own implementation that monitors references and memory block sizes, etc.
- *Rational Purify*[5] is a commercial memory debugger for Linux, Solaris, and Windows.

[4] See `http://valgrind.org/`
[5] See `http://unicomsi.com/products/purifyplus/`

8.6 Undefined Behaviour

Buffer overflows are a form of *undefined behaviour*: something that a program does that causes its future behaviour to be unknown. It might continue working or it might do something totally unpredictable, such as executing attacker-supplied code in the case of a successful remote code execution attack.

Programming languages like C define many forms of undefined behaviour, besides buffer overflows. Examples of undefined behaviour in C include:

- Dividing by 0
- Dereferencing a NULL pointer
- Overflowing a signed integer
- Underflowing a signed integer
- plus many more.

Testing for undefined behaviour, as with buffer overflows, requires special tool support. This is because the behaviour a program exhibits when performing an operation involving undefined behaviour cannot be relied upon. Compilers use this fact when optimising programs which can sometimes lead to very surprising results.

As an example, consider the following program `signed_overflow.c`:

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>

int main(const int argc, const char * argv[]){
  int32_t x, y;
  if (argc < 3){
    fprintf(stderr,"Usage: %s num1 num2\n",argv[0]);
    exit(1);
  }

  x = atoi(argv[1]);
  y = atoi(argv[2]);

  if (y <= 0){
    return 0;
  }

  if (x + y < x){
    printf("Overflow happens!\n");
  }

  return 0;
}
```

It reads two integers x and y given as command line arguments. If the second one, y, is positive, it then tries to test whether adding them together would cause overflow. It does so by checking whether $x + y$ is less than x. Given that y is known to be positive, this should happen only if overflow occurs when they're added together.

The program behaves as expected when compiled (under Clang 7.3.0) with no optimisation:

```
$ clang -O0 signed_overflow.c -o signed_overflow
$ ./signed_overflow 2147483647 10
```

```
Overflow happens!
$
```

Let's see what happens, however, when compiled with even the most mild optimisation level:

```
$ clang -O1 signed_overflow.c -o signed_overflow
$ ./signed_overflow 2147483647 10
$
```

Surprisingly, the program *doesn't* print "Overflow happens!".

While this behaviour might seem odd, the program has been legally compiled: signed overflow is undefined behaviour. Therefore, in any situation where the original program is guaranteed to perform undefined behaviour, the compiler is allowed to make it do anything it wants. In this case, the compiler has chosen to have the program do the thing that speeds up the program the most. Specifically, it has chosen to entirely omit the section of the program that does:

```
if (x + y < x){
  printf("Overflow happens!\n");
}
```

Whenever this bit of code doesn't perform undefined behaviour (i.e. doesn't cause signed overflow), it is equivalent to doing nothing. On the other hand, whenever it does perform undefined behaviour, the compiler is allowed to have the program do whatever it wants. Thus removing this section of code is entirely legal: in all cases where undefined behaviour doesn't occur the program behaves identically.

In general, a compiler is allowed to perform any transformation on a program so long as it ensures that the program's behaviour remains unchanged in all cases where it doesn't exhibit undefined behaviour.

Bugs similar to the one in the program above have led to vulnerabilities in Linux kernel drivers when the compiler has optimised away an overflow test of the above form that was being used to guard against accessing a buffer out-of-bounds.

Numerous articles exist online that detail how to rewrite overflow checks like the one in the faulty program above to ensure that they correctly check for over or underflow while avoiding undefined behaviour.

The unpredictability of how a program will behave when performing undefined behaviour means that undefined behaviour is important to test for. However, for the same reason, testing for undefined behaviour requires special help from the compiler. Essentially, the compiler produces a program that includes checks to ensure that execution is halted with an error whenever undefined behaviour is about to occur.

As with memory debuggers, these checks necessarily incur some performance overhead.

Tool Support

The Clang C compiler provides options for compiling code to enable it to be tested for undefined behaviour. With these options turned on, Clang builds programs so that they generate an error whenever they perform undefined behaviour, as shown in the following example.

```
$ clang signed_overflow.c -o signed_overflow \
        -fsanitize=undefined-trap -fsanitize-undefined-trap-on-error
$ ./signed_overflow 2147483647 10
Illegal instruction: 4
$
```

Here we see that the `Illegal instruction` error is generated when signed overflow occurs.

8.7 Confidentiality

Data security is often characterised in terms of three properties:

Integrity: prevention of unauthorised data modification.

Confidentiality: prevention of unauthorised data disclosure.

Availability: ensuring attackers cannot deny legitimate access to data.

The kinds of security vulnerabilities we have been considering so far involve violations of *integrity* or *availability*:

- SQL injection violates the integrity of the database interface.
- Buffer overflow attacks violate memory integrity and may cause program crashes, affecting availability.

Fuzzing can be an effective way of finding such vulnerabilities by looking for when integrity or availability violations occur, e.g. when a program crashes due to a buffer overflow. This is because these kinds of violations are *directly observable*. However, the same is not always true for confidentiality violations and, for this reason, using testing to find such vulnerabilities is much harder.

8.7.1 Overt Channels

The first kind of confidentiality vulnerability we will consider is the simplest, namely unwanted *overt channels*. An overt channel is a mechanism (whether intended or not) for directly transferring data in a system.

The following C program `overread.c` contains an unwanted overt channel, modelled loosely on the infamous Heartbleed bug in OpenSSL.

```
#include <stdlib.h>
#include <stdio.h>

int main(const int argc, const char * argv[]){
  if (argc < 3){
    fprintf(stderr,"Usage: %s strlen string\n",argv[0]);
    exit(1);
  }

  const unsigned long slen = strtoul(argv[1],NULL,10);

  for (unsigned long i=0; i<slen; i++){
    putchar(argv[2][i]);
  }
  putchar('\n');

  return 0;
}
```

Its job is to take two arguments: the first is a number n while the second is a string whose length is assumed to be not less than n. The program's job is to simply print back the first n characters of the string it was given.

However, if given a number n larger than the length of the given string, then it will print back extra data, as the following example invocation shows:

```
$ ./buffer_overread 80 test
testGHC_DOT_APP=/Applications/ghc-7.8.4.appTERM_PROGRAM=Apple_TerminalTERM=xt
```

When invoked with the argument n of 80 but with a string "test" whose length is just 4 characters, the program prints the 4-character string "test" but then goes on to print adjacent data from memory, which in this case contains the values of the program's environment variables. Were sensitive data sitting in memory near the string "test", then this program could conceivably disclose it given an appropriate argument n. A vulnerability similar in nature to this one existed in the OpenSSL SSL/TLS library, which could be exploited to learn the private keys in use by vulnerable web servers as well as users' session cookies and passwords, which were all stored in memory.

Overt channels exist when necessary access checks are missing. In the example above, the program fails to check that the argument n does not exceed the length of the given string. To test for the presence of overt channels, one can therefore test whether the program performs the necessary checks. However, this requires knowing what checks the program should be performing, which is not always obvious. The Heartbleed bug in OpenSSL was due to missing access checks of this kind yet lay undiscovered for years, despite it being one of the most highly used libraries of its kind.

8.7.2 Covert Channels

Covert channels are all those that are not overt ones. Such a channel allows indirect leakage of information. The following program is an example of such a channel. It references two variables: `secret`, which contains secret information; and `public`, which is observable to potential attackers. For instance, this function could be part of an operating system kernel implementing part of a system call being performed by one process, and `secret` might hold data pertaining to other processes in the system that are supposed to be kept confidential; `public` might hold the result returned to the process making the system call. While more complicated in nature than the following program, examples of covert storage channels are well known in operating system kernels that are otherwise supposed to be secure, and in general are very difficult to diagnose.

```
if (secret % 2){
   public = 1;
}else{
   public = 0;
}
```

This program leaks in the publicly-observable variable `public` the parity of the secret variable `secret`. This program is an example of a covert *storage channel*, since secret information is indirectly stored in a public place. On the other hand, the next program has an example of a covert *timing channel*:

```
while (secret) {
   secret--;
}
printf("Finished!\n");
```

Here, the time at which the publicly observable output "Finished!" becomes available depends on secret information, and so leaks that secret information to an attacker who can measure the passage of time. Timing channels are generally more difficult to diagnose than storage channels.

In general, overt channels are easiest to diagnose, while timing channels are the most difficult. The relative *bandwidth* of these kinds of channels is inversely related to their difficulty to diagnose: timing channels tend to have lower bandwidth than storage channels, which have lower bandwidth than overt channels.

Testing for Covert Channels

Testing for covert channels is very difficult, and the best ways to perform such testing are still an active area of research. Unlike overt channels, there are no missing access checks to test for. The only way to test for these kinds of channels is to observe how a program's public outputs or timing behaviour *changes* in response to changes to the data it is supposed to be keeping secret. For instance, in the first program above, which has a storage channel,

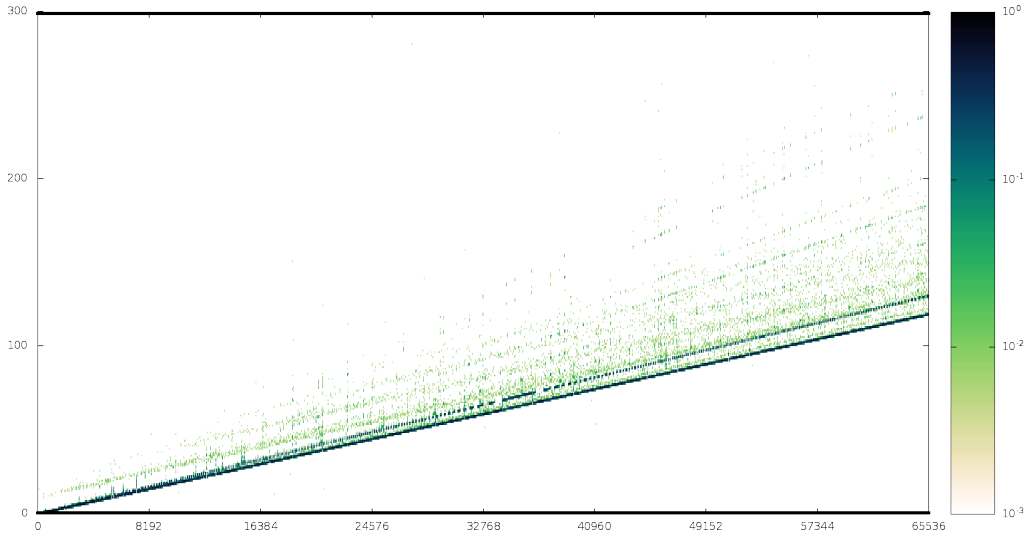
altering the parity of the variable `secret` causes the final value of public variable `public` to change. Similarly, altering the value of `secret` for the second program will affect the time at which it produces its public output.

There is evidence for the existence of a covert channel when, all other factors being held constant, the value of a public output (including the time at which the output is produced) changes in response to changes in some secret. In cases where we can enumerate *all* possible values of a secret and measure how some public output changes in response to changes in the secret, we can statistically characterise the *capacity* of the associated covert channel, that is how many bits of information it leaks about the secret in the public output.

Even when we can't run the program for all possible values of the secret, we can still calculate an upper bound for channel capacity from the number N of distinct observations that can be made about the public output. This upper bound is simply $log_2\ N$. For example, in the first program above, which has the storage channel, the public variable `public` is assigned either 0 or 1 (depending on the secret value `secret`), thus there are two possible observations, and so an upper bound for the channel capacity (i.e. for how any bits of information that `public` leaks about `secret`) is $log_2\ 2 = 1$. This makes sense: we know already that this program leaks the parity (i.e. leaks the least significant bit) of `secret`.

Tighter bounds on how much information a channel might leak can be computed by observing how the channel behaves for all possible values of the secret, summarised in a *channel matrix*. The channel matrix can be built by by running a large corpus of tests and observing how the public output changes as the secret is varied.

The channel matrix for the timing leak for the second program is depicted below. Values on the x-axis are different values for the (16-bit) secret `secret`, while values on the y-axis are different values for the running time of the `while`-loop as measured in milliseconds. The colour of each pixel $\{x, y\}$ denotes the probability of seeing the running-time y when the secret is x with darker colours denoting higher probability. Smaller values of the secret tend to produce smaller running times, and this pattern is stable across the entire matrix; however, there is also a bit of noise.



When the secret is zero, the running time also tends to be 0 milliseconds. We see also that the running time varies up to about 240 milliseconds. Thus an upper bound for the capacity of this channel (i.e. how many bits of the value `secret` are leaked in the loop's running-time as measured in milliseconds) would be around $log_2\ 240 \approx 7.9$.

Applying Shannon's information theory, however, we can compute a tighter bound from the channel matrix, which for the above matrix is ≈ 6.45 bits. This bound is tighter because it takes into account the probability of seeing each observation: while there are 240-odd different observations that can be made, some of them (such as all of those above about 110) almost never occur. Also there is a noticeable amount of noise: when observing a running time around 110 milliseconds, it might well indicate (the bottom dark line) that the secret's value is around 65000; however, it could also be around 57000 (the top dark line) or (with less probability) as low as around 27000.

To build the channel matrix above we simply measured the running time of the `while`-loop 300 times for each

of the 65535 possible values of `secret`. If we, say, observe that when the secret is 50 that the loop runs for 0 milliseconds 246 times but runs for 1 millisecond the other 54 times, then cell $\{50, 0\}$ of the matrix has value $\frac{246}{300}$ while cell $\{50, 1\}$ has value $\frac{54}{300}$ and all other cells $\{50, y\}$ for all other y have value 0. In general, if we run K trials for each value x of the secret and we observe that output y appears for this value of the secret $n_{x,y}$ times, then cell $\{x, y\}$ of the channel matrix has value $\frac{n_{x,y}}{K}$. The more trials K we run for each value of the secret, the better the estimate of the channel's capacity that we can calculate, because the values $\frac{n_{x,y}}{K}$ in each cell $\{x, y\}$ of the channel matrix become more accurate.

Tool Support

Standard algorithms exist for computing the channel capacity from the channel matrix, the most common of which is the Arimoto-Blahut algorithm. The channel matrix above was produced using the open source tools at: `http://ssrg.nicta.com.au/software/TS/channel_tools/`.

Currently there exist no tools or frameworks for automating this kind of testing, however. The inherent difficulty of testing for confidentiality violations means that the best known way to diagnose them currently is by using techniques based on formal (i.e. logical) reasoning about a program's semantics which, as mentioned in chapter 1, requires very specialised skills. At the time of writing, the best known example of this kind of reasoning is the proofs of information flow security for the seL4 kernel, which essentially prove the absence of storage channels in that microkernel.

Chapter 9

Automated test generation using symbolic execution

9.1 Introduction

One of the downfalls of standard testing is that, even if we run 1 million tests and have no failures, there is no guarantee that the next test we run will not fail. Recall Dijkstra's famous quote from Chapter 1:

> "*Program testing can be used to show the presence of bugs, but never to show their absence!*" — Edsger W. Dijkstra[1]

However, what if we *could* run every possible input on a program, even if there were an infinite number?

Symbolic execution is one approach to software verification that aims to do just that. The idea behind symbolic execution is that, instead of executing a program with concrete values, we execute it with *symbolic* values, which effectively execute multiple values at one time.

In this chapter, we will look at symbolic execution, and its variant *dynamic* symbolic execution, and how they can be used to verify software.

9.2 Symbolic execution

Early symbolic execution

Although first proposed by James King in 1976[2], symbolic execution is still in its infancy, and has only recently emerged as a realistic way to verify software.

While early work by researchers such as King were promising, symbolic execution suffered from two major problems:

1. Computers were small with limited memory and slow processors, compared to today's machines.
2. The constraint solving tools required for symbolic execution were limited.

[1] In *Notes on Structured Programming*.
[2] J. C. King, "Symbolic execution and program testing". *Communications of the ACM*, 19(7):(385–394), 1976.

Modern computers are significantly faster than those in the 70s, and memory is cheap. Further, constraint solving technology has improved by several orders of magnitude since the later 1970s, meaning that symbolic execution has become a hot topic of research again.

Constraint solving

A key technology required for symbolic execution is *constraint solving*. Constraint solving is an approach to generating solutions for mathematical problems expressed as constraints over a set of objects.

Each constraint to be solved has three parts: (1) a set of variables; (2) a *domain* for each variable (e.g. the integers, characters, or real numbers); and (3) the set of the constraints.

The easiest way to understand constraint solving problems is via an example. Consider the following constraint over the set of integers and variables x and y:

$$x \leq 0,\ y > 0,\ x + y > 100 \tag{9.1}$$

Constraint solvers can be used to answer three types of questions:

1. Is there a solution to this problem?
2. If "yes" to 1, what is a solution to the problem?
3. Is one constraint entailed by another?

There are, of course, many solutions to the constraint in Equation 9.1 so a constraint solver will answer "yes" to the first question. If asked to provide a solution to the constraint, it could return a number of answers, such as $x = 0,\ y = 101$, or $x = -10000,\ y = 200000$ (although the former more likely in most constraint systems!).

An example of a constraint that is not solvable is:

$$x < 0,\ y < 0,\ x + y > 0 \tag{9.2}$$

If x and y are both negative, then their sum cannot be positive. As such, a constraint solver will answer that this is *unsatisfiable*.

For checking constraint entailment, we ask the solver if constraint C *entails* constraint D, written $C \vdash D$. Entailment is similar to logical implication, and $C \vdash D$ is asking: is any solution for C also a valid solution for constraint D. In essence, if C stronger than D? That is, if everything on the left of $\vdash$ is true, does that mean that everything on the right is also true?

As an example, the entailment $x > 0 \vdash x \geq 0$ holds: if x is strictly positive, then it is also weakly positive. That is, any value of x that satisfies $x > 0$ also satisfied $x \geq 0$.

However, the inverse, $x \geq 0 \vdash x > 0$, does not hold, because $x = 0$ is a solution for the former but not the latter.

Constraint solvers can also answer more complicated problems, such as:

$$x < 200, y > 200, z = x - 100 \quad \vdash \quad y \geq 2z$$

A constraint solver should infer that $z < 100$, and therefore $2z < 200$; so because $y > 200$, it must always by the case that $y \geq 2z$.

Symbolic states

Constraints, such as those used in Equation 9.1 can be used to assign *symbolic* values to program variables. By symbolic, we mean values that represent possible more than one value. For example, the constraint $x < 0$ can be

a symbolic value for a variable x, indicating that x is negative. This is in contrast to a *concrete* value of x, such as $x = 1$, which defines a specific value for x.

When we perform standard software testing, we provide a program with concrete inputs, and those inputs are executed. Exploring the entire input space with concrete values is impossible for even the most trivial examples.

At any point during execution of a program, the program will have a *state*, which is a mapping from all variables to their concrete values at that time. As a program executes, calculations (instructions) are executed on the variables in that state, and their values are updated. For example, the concrete state may be $x = 5, y = 10$, and the instruction $x := y + 1$ (assignment) will update the state to $x = 11, y = 10$.

The key idea behind symbolic execution is to replace the concrete state with a *symbolic state*, in which the variables map to symbolic values, and to perform calculations on those symbolic values. By doing this, a program can be executed for many inputs at one time.

Some examples

Example 68 As an example, consider the example program in Figure 9.1, which swaps the values of two integers, `x` and `y`, if and only if `x` is less than `y`. The result is that `x` is always greater than `y`.

```
1.  void swap(int x, int y)
2.  {
3.    if (x < y) {
4.      x = x + y;
5.      y = x - y;
6.      x = x - y;
7.    }
8.    assert(x >= y);
9.  }
```

Figure 9.1: A program for swapping two integers.

We can execute this program on a concrete state, providing concrete values for `x` and `y`; for example, `x = 4` and `y = 10`. The concrete state of the program would be updated as follows:

Line	Program fragment	Variable values		Notes
		`x`	`y`	
1.	`void swap(int x, int y)`	4	10	
3.	`if (x < y)`	4	10	(branch true)
4.	`x = x + y`	14	10	
5.	`y = x - y`	14	4	
6.	`x = x - y`	10	4	
8.	`assert(x >= y)`	10	4	(assertion true)

Executing the program for this pair of inputs will execute one path out of two, but for just one possible execution of that path. Any input in which `x < y` will also execute that path. If we are testing, we would (hopefully) select a value such that `x >= y`, covering the other path. In both cases, the assertion at the end of the function will be true.

However, if we replace the concrete state with a symbolic state, starting with the variable `x` and `y` being completely unconstrained, we can cover every possible input by symbolically executing just the two paths. At the branch, we split the symbolic execution into the two paths, executing all inputs on both paths by executing all instructions symbolically. At each point in time, the value of the variables is represented as a constraint, called the *path constraint*, which summarises all possible values of the program variables at that point.

The following illustrates how the symbolic state is modified during execution. In this illustration, we use capitalised variable names, `X` and `Y`, to represent the symbolic values of `x` and `y` at the start of the execution:

Line	Program fragment	Path constraint		Other
		Path `x < y`	Path `x >= y`	
1.	`void swap(int x, int y)`	`x=X, y=Y`		(Only one path so far)
3.	`if (x < y)`	`x=X, y=Y, X<Y`	`x=X, y=Y, X>=Y`	(Fork into two paths)
4.	`x = x + y`	`x=X+Y, y=Y, X<Y`	–	
5.	`y = x - y`	`x=X+Y, y=X, X<Y`	–	
6.	`x = x - y`	`x=Y, y=X, X<Y`	–	
8.	`assert(x >= y)`	`x=Y, y=X, X<Y`	`x=X, y=Y, X>=Y`	(assertion true)

At line 5, the assignment `y = x - y` means that `y=(X+Y)-Y`, which simplifies to just `y=X`. A similar simplification occurs at line 6. The constraint solvers are used to simplify these expressions.

From the symbolic execution above, we can see that the assertion `x >= y` holds at the end of both parts. Because symbolic values are used instead of concrete values, both of the paths are executed for all possible values of `x` and `y`. Therefore, we have executed the program for all possible inputs, and shown that the assertion at line 8 will always be true; that is, there is no test input that will violate it. We can show this by taking the path constraints at the assert statement and checking if they entail the assertion. The entailments `x=Y, y=X, X<Y` $\vdash$ `x >= y` and `x=X, y=Y, X>=Y` $\vdash$ `x >= y` both hold, so the assertion always holds.

If the programmer had made a mistake on line 6 and used addition instead of subtraction, the final constraint for the first path would be: `x = (X+Y) + Y`, and the assertion would be violated.

Example 69 As a more complicated example, consider the following function[3] written in C, which allocates memory to a pointer:

```
 1. int* func(int x; int y)
 2. {
 3.   int* p = 0;
 4.   int s = x + y;
 5.   if (s != 0) {
 6.     p = malloc(s);
 7.   }
 8.   else if (y == 0) {
 9.     p = malloc(x);
10.   }
11.   return p;
12. }
```

A common property for symbolic execution tools are used to check is whether functions can return null. Let's assume then that we want to analyse whether this function can return `null` (that is, the value of `p` is 0).

Figure 9.2 shows the symbolic execution tree for this program. There are four possible paths through the program, corresponding the the four variations of true/false at the branch points. The constraints `x=X` and `y=Y` are in every node, and have been omitted for readability.

To check whether the pointer can possibly be null, we have to check whether for at least one path, whether the final path constraint conjoined with the constraint `p=0` is satisfiable. If it is satisfiable, there is at least one execution of the program that results in a null pointer.

From the execution tree, it is clear that the path on the right is one such path, with `p=0` being part of the path constraint.

A real fault Symbolic execution can find considerably more subtle faults than the one above. For example, consider the following program fragment from the GNU CoreUtils package.

In this case, the program may be asked to "untabify" some input, meaning to remove tabs and replace them with

[3] Adapted from the example at `http://martinsprogrammingblog.blogspot.com.au/2011/11/symbolic-execution.html`

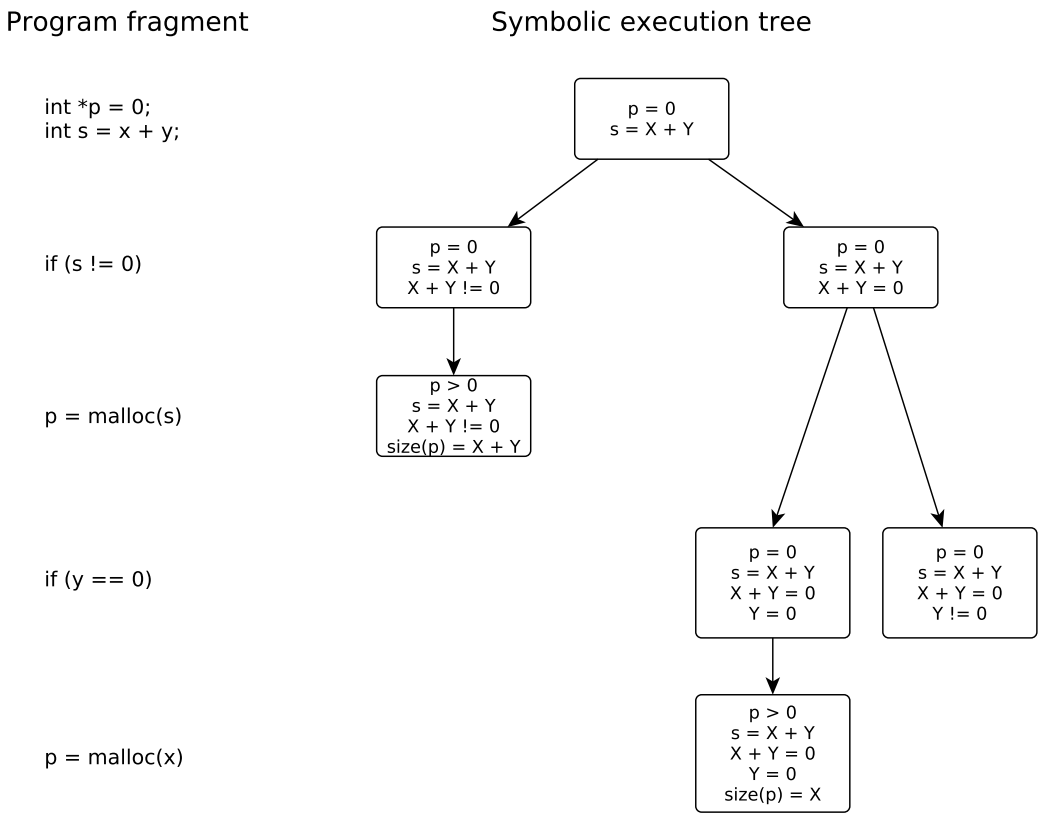


Figure 9.2: Execution tree for the function `func`

```
1322:     s = xmalloc(MAX(8, chars_per_input_tab));
....
2655:     width = chars_per_c - (input_position % chars_per_c);
2666:
2667:     if (untabify_input)
2668:     {
2669:       for (i = width; i; --i)
2670:         *s++ = ' ';
2671:       chars = width;
2672:     }
```

spaces. The variable `width` is used to calculate the amount of spaces that need to be inserted. The programmer has made the incorrect assumption that `width` is always between 0 and `chars_per_c - 1`. When `input_position` is positive, this assumption holds; however, because backspaces are permitted in the input string, `input_position` can be negative, meaning that `width` can become larger than the size of `s`, and an overflow can occur at line 2670, as spaces are added past the end of the string.

This subtle fault was found by the KLEE symbolic execution tool[4] in 2008, and had been in the GNU CoreUtils package at least since it was first uploaded onto a CVS repository in 1992, despite the CoreUtils package having an unusually comprehensive test suite, and having been used for many years.

Such subtle faults are difficult to detect using dynamic software testing, and are where symbolic execution can be

[4]See the following paper for details on KLEE: C. Cadar, D. Dunbar, and D. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." *OSDI*. Vol. 8. 2008.

useful.

Symbolic test oracles

Symbolic execution is only a way to explore test inputs for a program, but it cannot tell us whether that program meets its requirements. For this, we need the equivalent of a test oracle for symbolic execution. In the example above, we use assertions as test oracle; however, most programs do not contain useful assertions that can be used.

Standard test oracle solutions, discussed in Chapter 5, cannot necessarily be applied directly, because they are designed for concrete examples, not symbolic examples.

Despite this, some of these solutions would work well. For example, if we have a golden program, then we can execute a path the program-under-test using symbolic execution, and compare the final path constraint to the path constraint generated when executing the same symbolic input on the golden program. Also, the idea of metamorphic oracles can be applied to symbolic execution: for some programs, a specific symbolic value may return a particular output.

In most cases, symbolic execution is used to look for *generic* faults, such as accessing arrays out of bounds, dividing by zero, returning null pointers, or certain security violations. These can be checked for any program using a generic "oracle", and empirical evaluation shows that they work very effectively. Cadar et al.[5] found many faults in the GNU CoreUtils package, such as the one discussed above, several of which had been present for over a decade.

Test input generation

Recall from earlier in this section that constraint solvers can be asked to provide a solution for a constraint. As such, an important part of symbolic execution is test input generation: test inputs can be generated by asking the constraint solver for a solution to the path constraint.

For example, consider the earlier program that allocated memory to the pointer `p`. We can ask for test inputs for each path, giving us tests that achieve path coverage for the program.

More importantly, if our symbolic execution tool finds that the program can indeed return a null pointer, then we can also ask for a *concrete* test input that produces this case. This provides programmers with a value concrete case for debugging, which are much easier for humans to reason about than symbolic traces.

In the example, the fourth path results in a null point (`p = 0`), and we can ask the constraint solver to generate inputs for `x` and `y` can be solving the path constraint in the final node of that path:

$$x = X, y = Y, p = 0, s = X + Y, X + Y = 0, Y! = 0$$

Thus, the constraint solver must generate values `X` and `Y` equal 0, but `Y` itself does not. Any values such that `X = -Y` will suffice, such as `X = 1` and `Y = -1`. Execute these inputs on the program will return a null pointer.

Limitations

There are several challenges related to the scalability of symbolic execution, limiting their uptake, but which also present some interesting research problems.

Constraint solving Symbolically executing all inputs on a path is expensive due to the cost of the underlying constraint solving. Current evaluations put the cost of symbolically executing a path at about 80 times that of concrete execution. This means that even a small set of unit tests covering a set of paths that take one minute to execute would take over an hour to symbolically execute.

There are two general solutions that significantly reduce the execution cost of symbolic execution:

[5] See the following paper for details on KLEE: C. Cadar, D. Dunbar, and D. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." *OSDI*. Vol. 8. 2008.

1. *Removing unnecessary constraints*: Each branch in a program generally depends on a small number of variables (1 or 2), so to solve a constraint at the branch (to fork the symbolic execution process), we can eliminate those constraints that are not relevant.

   For example, consider a symbolic execution tool encountering a branch `if (x < 0)`, with the path constraint $x + y > 0, z < 2a, y < 10$, and wants to check if it is feasible that `x` can be less than 0, the constraint $z < 2a$ can be removed from this check, as it will not influence the result.
2. *Caching solutions*: Branches in programs tend to have lots of constraints that are similar to each other. As such, when solutions are required; e.g. generating a test input; the symbolic execution tool can cache solutions are try to reuse them later.

The field of constraint solving is a very active research area, and much progress is being made to speed up how efficiently constraints can be solved; however, the problem will persist, as reasoning over an entire collection of inputs will always take much longer than a single input.

Path explosion The examples that we have seen so far are toy examples used to illustrate how symbolic execution works. If we want our symbolic execution tool to explore all possible paths through a program, the resulting number of paths explodes exponentially as the number of branches in the program increases.

For example, consider the control-flow graph in Figure 9.3 from a real program that is non-trivial, but not unusually large. Exhaustively symbolically executing every path through this program is not feasible.

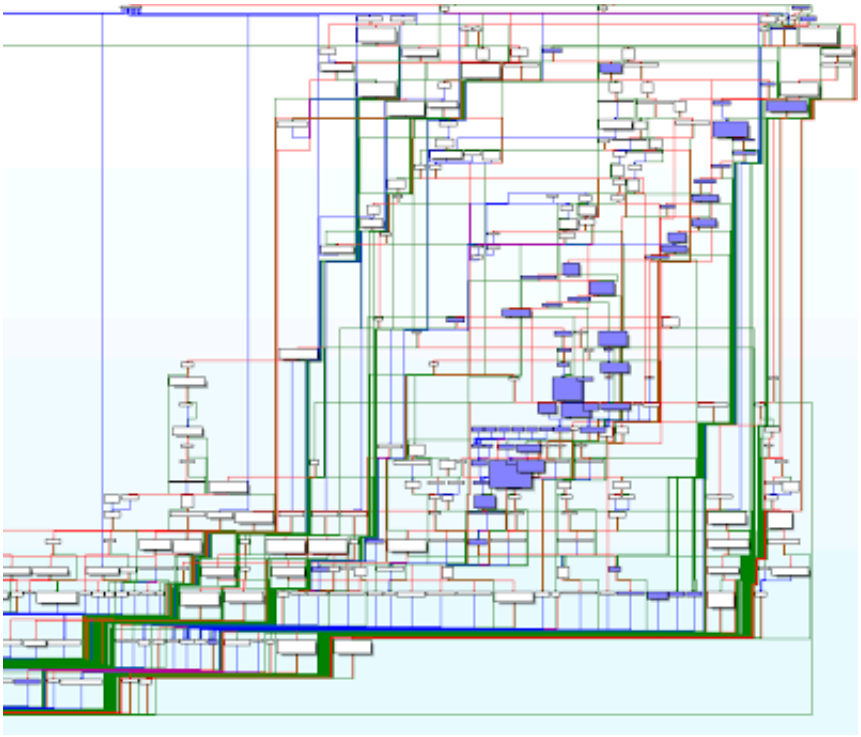


Figure 9.3: A control-flow graph for a non-trivial program.

Much research has gone into mitigating the path explosion problem, sacrificing completeness for scalability. Most of this research looks heuristics for executing only a small subset of possible paths while minimising the impact this has on fault finding. Some typical solutions are:

1. *Coverage-based search*, which iteratively chooses the path with the highest number of unexplored statements/branches to achieve branch coverage rather than path coverage, or the paths with code that was just added to the program.
2. *Random path search*, which works surprisingly well, but not as systematically as coverage-based search.

Loops! In our examples, the programs contained no loops. Loops further exacerbates the path explosion problem by giving us a potentially infinite number of paths. Further to this, while we can use testing heuristics, such as the 0-1-many rule to select a small set of paths from a loop, executing certain paths after a loop may require the loop to execute a specific number of times. The automatic search algorithms employed by symbolic execution tools are not intelligent enough to determine these, and proceed rather blindly, resulting in the tool getting "stuck".

Unsolvable paths A final issue is with unsolvable paths: those paths that cannot be symbolically executed, effectively halting exploration of the path. Paths are typically unsolvable for two reasons:

1. *Unsolvable constraints*: a branch or instruction contains a statement that is outside of the scope of the constraint solver. For example, many state-of-the-art constraint solvers consider only linear constraints (e.g. of the form $z \geq 2x + 4y$) over integers, meaning that they can handle only integers and sometimes arrays[6]. So, when encountering a program with real numbers or nonlinear constraints, the symbolic execution will fail at a point that these are used, or must revert to approximate evaluation, which may be unsound.
2. *External calls*: if a program contains a call to an external function, such as a library, or a function that reads user input, then it cannot reason about this symbolically, as it does not have access to the source code. As such, the symbolic execution algorithm must either halt, or approximate what the answer will be for the output.

   The KLEE symbolic execution tool has its own model of the Linux system library, which it uses to get around this problem; however, this is not a generalisable solution.

Tools

There are a handful of mature symbolic execution tools, primarily built from research prototypes.

Java PathFinder[7] is a model checker and symbolic execution tool for Java, built and maintained at the NASA Ames Research Centre.

KLEE[8] is the most mature symbolic execution tool for LLVM (Low Level Virtual Machine), and many research projects have built on KLEE since its release in 2008. KLEE is still actively maintained.

Sypy[9] is a symbolic execution tool for Python, which has not been systematically evaluated. It appears that the development on the tool has halted.

9.3 Dynamic symbolic execution

Dynamic symbolic execution (DSE) is a test input generation technique that is a cross between random testing (Section 7.3) and symbolic execution, with the aim of overcoming the weaknesses of each of these.

It is perhaps more accurately described as a combination of dynamic execution and symbolic execution. Essentially, a DSE tool runs a concrete input on a program, and simultaneously performs symbolic execution on the path that this input executes. It then uses the path constraint generated by the symbolic execution to generate a new test.

Overview

DSE is used for dynamic test input generation. It uses symbolic execution, but the approach to test input generation is different to that of symbolic execution.

[6] Note that characters can be treated as integers, and strings can be treated as arrays of characters, so this is sufficient for many programs.
[7] `http://babelfish.arc.nasa.gov/trac/jpf`
[8] `http://klee.github.io/klee/`
[9] `https://github.com/diffoperator/Sypy`

Example 70 We illustrate this with an example. Consider the following program, with a fault at the line that says "abort()":

```
1. void good_bad(char s[4]) {
2.   int count = 0;
3.   if (s[0] == 'b') count++;
4.   if (s[1] == 'a') count++;
5.   if (s[2] == 'd') count++;
6.   if (s[3] == '!') count++;
7.   if (count > 3) abort();  //fault
8. }
```

This program will only fail on the input "bad!". Using random testing, random arrays of four characters will be generated, but encountering the fault at line 7 is highly unlikely. If we consider a 32-bit architecture, the probability of randomly generating the string "bad!" requires us to randomly choose 'b' at index 0 (probability of 1 in 2^{32}, 'a' at location 1 (probability of 1 in 2^{32}), etc., leaving us with the probability of 1 in 2^{128} that the fault will be encountered.

If we use dynamic symbolic execution on that program, we first generate a random input, and symbolically execute the path on that input while also executing the concrete input. For illustration's sake, let's consider that the randomly generated input is the string "good". The path constraint will be: $s_0 \neq$ `b`, $s_1 \neq$ `a`, $s_2 \neq$ `d`, $s_3 \neq$ `!`, resulting in `count` being 0, and the fault not being executed.

Next, we use a constraint solver to generate a *new* test. With symbolic execution, we would use the path constraint to generate a concrete input for the path, however, we have already executed a concrete input for that path: the input "good". So instead, we generate a new test by *negating* one of the branches in the path constraint; e.g. the final branch: $s_0 \neq$ `b`, $s_1 \neq$ `a`, $s_2 \neq$ `d`, $s_3 =$ `!`. This will generate a new input, such as "goo!".

We can systematically repeat this for all 16 combinations of branches, symbolically executing each time, which will provide us with complete path coverage. One of the 16 combinations will be the constraint $s_0 =$ `b`, $s_1 =$ `a`, $s_2 =$ `d`, $s_3 =$ `!`, which generates the input "bad!", and will execute the fault.

Example 71 The above example illustrates how DSE differs from symbolic execution, but does not illustrate why using concrete values can be advantageous. For this, consider the following example, which contains a call to an external function called `hash(char s[8])`, which calculates a hash value for a string:

```
1.  void hash_example(int x, char s[8])
2.  {
3.    if (x == hash(s)) {
4.      //do something good
5.    }
6.  }
```

In this example, the input `x` must be equal to the hash value of `s` to execute the code block inside. Such code is common in applications with security restrictions, and to test most of the application's functionality, the branch needs to evaluate to true for most tests.

Now, consider a case in which we cannot symbolically execute `hash`; either because we do not have the source code (it is an external function), or its logic is too complex for the constraint solver being used. If we used random testing, we are unlikely to execute a test such that the branch is true. If we used symbolic execution, the execution would halt at the branch.

However, if we use DSE, we can overcome this problem. First, the DSE algorithm will pick random values for `x` and `s`; for example, `x = 1019` and `s = ''s3fpmi''`. When the branch is symbolically evaluated, we cannot evaluate `hash(y)`, so we instead record the concrete value; e.g. 74233. Next, we add the concrete value to the path constraint instead of the symbolic value, to get the condition $x \neq 74223$.

To generate a new test, we negate this path constraint ($x = 74223$), solve this constraint, and get the input `x = 74223`, and the branch will evaluate to true, executing the code block instead the if statement. Brilliant!

Of course, if *both* sides of the equality are not symbolically executable (e.g. `hash(x) == hash(y)`), this will not work; but this solution works for a large number of cases in which we can access the concrete values but not the symbolic values of function calls.

In summary, the difference between symbolic execution and DSE is two things:

1. The concrete input is executed simultaneously with the symbolic input, instead of generating the test after symbolically executing a path. The first input is chosen arbitrarily (perhaps randomly).
2. Path constraints are modified to generate a *new* test after each path is executed.

Advantages

Advantages over random testing Clearly, the advantage of DSE over random testing is that it is not so random. While the first test case is random, using the program's control-flow to explore the input space provides a much better coverage of the program structure than with random testing.

Advantages over symbolic execution Using DSE to generate test inputs has a major advantage over using just symbolic execution: when an unsolvable path (e.g. unsolvable constraint or external function call) is encountered, DSE uses the concrete values for those variables that cannot be solved, and adds these to the symbolic state. This effectively under-approximates the symbolic values of those variables, but it allows execution to continue.

A second advantage is that it opens up to new search strategies to deal with the path explosion problem. For example, for their SAGE DSE tool[10], Microsoft researchers proposed a technique known as *generational* search. In generational search, one symbolically executed path is used to generate multiple new tests.

For example, in our good/bad example above, instead of generating one new test from the path constraint $s_0 \neq \texttt{b}$, $s_1 \neq \texttt{a}$, $s_2 \neq \texttt{d}$, $s_3 \neq \texttt{!}$, and executing this using DSE, SAGE generates four new inputs by negating *each* of the constraints in the path constraint, giving four new path constraints:

(1)	$s_0 = \texttt{b}$, $s_1 \neq \texttt{a}$, $s_2 \neq \texttt{d}$, $s_3 \neq \texttt{!}$	$\Rightarrow$	"bood"
(2)	$s_0 \neq \texttt{b}$, $s_1 = \texttt{a}$, $s_2 \neq \texttt{d}$, $s_3 \neq \texttt{!}$	$\Rightarrow$	"gaod"
(3)	$s_0 \neq \texttt{b}$, $s_1 \neq \texttt{a}$, $s_2 = \texttt{d}$, $s_3 \neq \texttt{!}$	$\Rightarrow$	"godd"
(4)	$s_0 \neq \texttt{b}$, $s_1 \neq \texttt{a}$, $s_2 \neq \texttt{d}$, $s_3 = \texttt{!}$	$\Rightarrow$	"goo!"

This gives four new inputs having executed only one symbolic path, reducing the cost of DSE significantly. As these four tests are executed, the next "generation" is born, which negates the constraints in the path constraints that have not been executed previously. For example, the test "bood" will generate three more constraint that result in the test inputs: (1) "baod"; (2) "bodd"; and (3) "boo!". The test "good" is not generated again because that path has already been covered. Another generation is executed, until we get the test "bad!".

In total, only 6 paths are symbolically executed, rather than 16. This generational search algorithm allows DSE to scale much better than using a more complete algorithm, such as depth-first or breadth-first search.

Limitations

DSE still suffers from all of the same limitations as symbolic execution: path explosion, expensive constraint solving, loops, and unsolvable paths. However, the use of concrete information during execution helps to mitigate all of these at some level, significantly improving the scalability.

However, DSE suffers from a problem that is not found in symbolic execution: *divergence*. Because a concrete input is used to drive which path is executed next, a DSE algorithm does not fork at each branch, but instead just collects the symbolic values of the path being executed. However, if a new test is generated and the path it executes is different to the path we expected, we say that a *divergence* has occurred. This is caused by the limitations in the constraint solvers used.

[10] See `http://research.microsoft.com/en-us/um/people/pg/public_psfiles/sage-in-one-slide.pdf`

For example, consider the following program:

```
 1.  void divergent(int x, int y)
 2.  {
 3.    int s[4];
 4.    s[0] = x;
 5.    s[1] = 0;
 6.    s[2] = 1;
 6.    s[3] = 2;
 7.    if (s[x] = s[y] + 2) {
 8.      abort(); //error
 9.    }
10.  }
```

If our first randomly generated input is `x = 0` and `y = 1`, then the branch statement at line 7 will be false: `s[x] = 0` and `s[y] + 2 = 2`. Because constraint solvers are not sophisticated enough to reason about symbolic addresses (or arrays) completely, the concrete values of `s[0]` (0 is the concrete value of `x`) and `s[1]` (1 is the concrete value of `y`) are used instead. The symbolic value of `s[0]` is `X`, and for `s[1]` it is 0. Because `x == 0 + 2` is false, the path constraint generated is $x \neq 2$. Negating this results in the new test inputs `x = 2, y = 1`. However, because the value of `s[x]` is dependent on the input, we get the values `s[x] = 1` and `s[y] + 2 = 2`, so the branch at line 6 again evaluates to false, and the fault is missed. The DSE algorithm will then terminate, and will miss the case `x = 3, y = 1`, which finds the fault.

In this case, concretisation results in divergence, which would not happen using symbolic execution. However, considering that the constraint solver is not able to reason about symbolic addresses, the symbolic execution could not continue at this point, so using concretisation will help in some cases.

Recent research studies how to analyse such problems with arrays and pointer arithmetic to provide better solutions, but these do not always work due to the complicated nature of pointers.

Tools

The most successful DSE tool to date is Microsoft SAGE[11]. SAGE is not available for download, however, it has been used successfully internally at Microsoft since 2008, finding over one third of all security vulnerabilities found in Windows 7. Microsoft currently host a lab of over 200 machines dedicated to running SAGE over their development code.

S^2E[12] is an open-source DSE tool for the LLVM, built on KLEE, which contains a sophisticated approach for generating sequences of method calls to find vulnerabilities in programs.

CREST[13] and jCUTE[14] are open-source DSE tools for C and Java respectively, primarily designed to allow researchers to modify and experiment with DSE.

9.4 Into the future...

Symbolic analysis, and dynamic symbolic execution in particular, are likely to play a significant role in testing in the future, as the complexity of software systems increase to the point where manual test generation is simply not feasible to achieve high coverage.

As constraint solvers continue to improve, the effect is passed on to symbolic execution tools who build on these. Constraint solvers are improving at a remarkable rate as researchers find new tricks to improve reasoning without

[11] `http://research.microsoft.com/en-us/um/people/pg/public_psfiles/sage-in-one-slide.pdf`
[12] `http://s2e.epfl.ch.`
[13] `http://jburnim.github.io/crest/`
[14] `http://osl.cs.illinois.edu/software/jcute/`

the loss of precision, and find heuristics to give approximate answers. Further, researchers are increasing the domains in which constraint solvers can find answers in reasonable times, including real numbers and certain nonlinear domains.

Symbolic execution and DSE are exciting research topics, with research being done in this department on those topics, as well as other symbolic reasoning approaches such as abstract interpretation.

Appendix A

A Brief Review of Some Probability Definitions

Probability theory is concerned with *chance*. Whenever there is an event or an activity where the outcome is uncertain, then probabilities are involved. We normally think of any activity or event with an uncertain outcome as an *experiment* whose outcome we observe. Probabilities are then measures of the likelihood of any of the possible outcomes.

An *experiment* represents an activity whose output is subject to chance (or variation). The output of the experiment is referred to as the *outcome* of the experiment. The set of all possible outcomes is called the *sample space*.

> **Definition 72** The *sample space* for an experiment is the set of all possible outcomes that might be observed.

Example 73 Some examples of experiments and their outcomes.

(i) An experiment involving the flipping of a coin has two possible outcomes Heads or Tails. The sample space is thus $\mathsf{S} = \{\mathsf{Heads},\ \mathsf{Tails}\}$.

(ii) An experiment involving testing a software system and counting the number of failures experienced after T = 1 hour has many possible outcomes: we may experience no failures, 1 failure, 2 failures, 3 failures, ... The sample space is thus $\mathbb{N} = \{0,\ 1,\ 2,\ 3, \ldots\}$.

(iii) An experiment involving the testing of a software system and recording the time at which the first failure occurs has a real valued sample space $\mathbb{R}$, although in practice this is more likely to be a time interval.

Probabilities are usually assigned to events.

> **Definition 74** Let S be a sample space. An *event* is a subset A of the sample space S, that is, $A \subset S$. An event is said to have *occurred* if any one of its elements is the outcome observed in an experiment.

The probability of an event A is a non-negative real number that relates to the number of times we observe an outcome in A. Often this real number is just the fraction of times that we observe an outcome in A over the total number of possible outcomes. We write:

$$P\{A\} = \lim \frac{m}{n}$$

where m is the number of outcomes in A and n is the total number of possible outcomes, that is, the number of elements in S. For two events, A and B, $A \cup B$ to refer to event A occurring or event B occurring, or both, and, $A \cap B$ to refer to event A and event B both occurring. Therefore $P\{A \cap B\}$ is the probability of events A and B both occurring. The *conditional probability* of an event A is the probability of A given that B has occurred. This is written $P\{A|B\}$. In conditional probabilities B is often called the *conditioning event*.

Probabilities must satisfy certain *probability laws* in order to be meaningful measures of likelihood. The following probability laws hold for any event A and any state space S:

(i) $0 \leq P\{A\} \leq 1$; and

(ii) $P\{\neg A\} = 1 - P\{A\}$

(iii) $P(S) = 1$.

That is: (i) the probability of an event occurring is between 0 and 1 inclusive; (ii) the probability of A not occurring is 1 minus the probability that it will; and (iii) the sum of the probabilities of all events in the state space is 1, and no events outside the sample space can occur.

Two events are said to be *independent* if the occurrence of one event does not depend on the other and the converse. For example, if we have two dice, the probability of one falling on the outcome 6 is independent of the outcome of the other dice. However, if we throw both dice, but one falls on the floor out of sight, while the other shows a 3, then the probability of their total equaling 7 is dependent on the probability of the second dice.

If events A and B are *dependent*, then the following laws hold:

$$P\{A \cap B\} = P\{A|B\}P\{B\} \quad \textit{Multiplicative Law}$$
$$P\{A|B\} = \frac{P\{A \cap B\}}{P\{B\}} \quad \textit{Conditional Probability}$$

If events A and B are *independent* then the following laws hold:

$$P\{A \cap B\} = P\{A\}P\{B\} \quad \textit{Multiplicative Law}$$
$$P\{A|B\} = P\{A\} \quad \textit{Conditional Probability}$$

The independent and dependent cases are related. For example, if A and B are independent, then $P\{A \cap B\}$ = $P\{A\}P\{B\}$, and therefore $P\{A|B\}$ is equal to $(P\{A\}P\{B\})/P\{B\}$ (the first conditional probability law), which is clearly $P\{A\}$ (the second conditional probability law).

Two events are said to be *mutually exclusive* if they cannot both occur. For example, if we throw a single die, then it is not possible that both 1 and 2 will result. If events A and B are not mutually exclusive, then the following law holds:

$$P\{A \cup B\} = P\{A\} + P\{B\} - P\{A \cap B\} \quad \textit{Additive Law}$$

If events A and B are mutually exclusive, then the following laws hold:

$$P\{A \cup B\} = P\{A\} + P\{B\} \quad \textit{Additive Law}$$
$$P\{A \cap B\} = 0 \quad \textit{Mutual Exclusion}$$

These laws are also related. If A and B are mutually exclusive, then $P\{A \cap B\}$, and therefore $P\{A\} + P\{B\} - P\{A \cap B\}$ (the first additive law) is $P\{A\} + P\{B\} - 0$, which is equivalent to the second additive law.

Random Variables

Now, suppose that we can represent each element of the sample space by a number. If we perform an experiment then for each element E of the sample space S, there is a certain probability that we will observe E as the outcome. A *random variable* assigns a number to each outcome in the sample space. Random variables that we will encounter in this subject are:

(i) Number of failures at time T, which is *discrete* a integer number. The sample space is T and the number that we assign to it as the random variable is the number of failures.

(ii) Time to first failure, which is a *continuous* real-valued number. The sample space is again T and the random variable in this case is the time τ to the first failure observed.

As an example, consider an experiment in which we execute random test cases on a piece of software for a period of 1 hour and observe the number of failures.

(i) The sample space that we will consider (and its not the only one possible either) is $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$.

(ii) Let X be a random variable that returns the number of failures experienced after 1 hour of testing (assuming we can suitably quantify failure).

(iii) Then we can ask questions such as what is the probability that there are fewer than N failures after 1 hour – written $P\{X<N\}$ – or what is the probability that we will have no failures after 1 hour – $P\{X\} = 0$. The random variable which we have called X is the number of failures experienced in 1 hour.

Probability Density Functions

Normally associated with each random variable is a *probability density function*, or sometimes a *probability law*, that assigns a probability to every outcome in the random variable's sample space. A random variable can be discrete or continuous.

- The sample space of a discrete random variable takes on specific values at discrete points $\{a_1, \ldots, a_k\}$.
- A continuous random variable takes on all values in the real line or an interval of the real line.

We think of the probability density function as a function f that maps each value of the sample space S to $[0, 1]$ so that $f : S \rightarrow [0, 1]$. For example, the probability density function of throwing a die is $f(o) = \frac{1}{6}$ for all outcomes o.

The two key properties that we require of probability density functions are as follows.

For Discrete Random Variables: we require that

(i) $f(X) \geq 0$ for all $x \in S$; and

(ii) $\sum_{x \in S} f(x) = 1$.

For Continuous Random Variables we require that

(i) $f(X) \geq 0$ for all $x \in S$; and

(ii) $\int_{-\infty}^{+\infty} f(x) = 1$.

The cumulative density function for a discrete random variable X is defined as

$$F_X(T) = \sum_{X \leq T} f(X)$$

that is, a sum of all of the probabilities for outcomes less than or equal to T. If we wish to calculate the probability $P\{A < X \leq B\}$ then this is simply

$$P\{A < X \leq B\} = F_X(B) - F_X(A).$$

For a continuous random variable X the cumulative density function is defined as

$$F_X(T) = P\{X \leq A\} = \int_{-\infty}^{A} f(X)$$

Probability in Reliability Measurement

Random variables and their distributions will be important to us because many of our reliability measures are expressed in terms of random variables and their distributions. Some examples are:

(i) Let T be a random variable representing the time of failure of a particular system. Then the *failure probability* is expressed as a cumulative density function:

$$F(t) = P\{T \leq t\}.$$

(ii) Reliability is simply the converse – it is the probability that a particular systems survives — that is, does not experience a failure — until after time t:

$$R(t) = 1 - F(t) = P\{T>t\}.$$

Appendix B

Maximum Likelihood Estimation

Maximum likelihood estimation is by far the better technique for estimating model parameters. Unfortunately it also often requires numerical solutions to solve sets of equations for those very parameters.

Intuitively the maximum likelihood estimator tries to pick values for the parameters of the basic execution time model that maximise the probability that we get the observed data. For example suppose that we had the failure intensity data shown in Figure B.1.

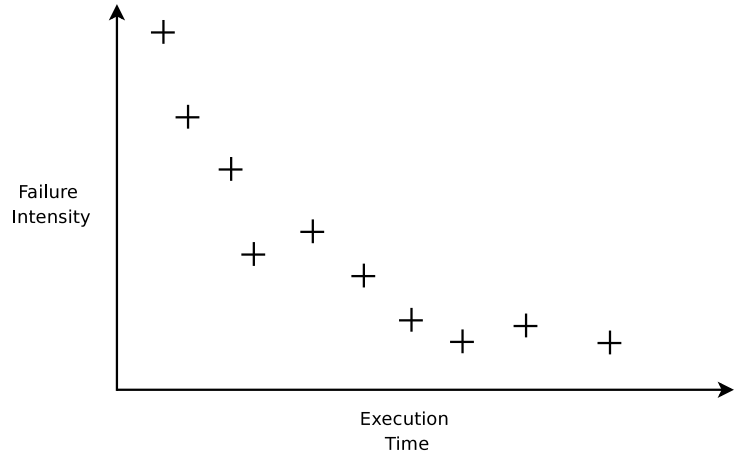


Figure B.1: Failure intensity against time derived from failure time data for a system.

Then we start exploring the parameter space consisting of pairs of values $(\lambda_0,\ \nu_0)$ for values that will maximise the likelihood of obtaining the observed values. When we start exploring the values of λ_0 and ν_0 we may find that the curves like something like those in Figure B.2.

To estimate the parameters, maximum likelihood now works as follows. Suppose that we have only one parameter θ instead of the two parameters in the Basic Execution time model. Now, if we make n observations x_1, x_2, …, x_n of the failure intensities for our program the probabilities are:

$$L(\theta) \ = \ P\{X(t_1) = x_1\}P\{X(t_2) = x_2\} \ldots P\{X(t_n) = x_n\} \tag{B.1}$$

To function $L(\theta)$ reaches its maximum when the derivative is 0, that is,

$$\frac{\mathrm{dL}(\theta)}{\mathrm{d}\theta} \ = \ 0 \tag{B.2}$$

For example, if we have an exponential probability law $\theta e^{-\theta T}$ with a parameter θ and we make n observations

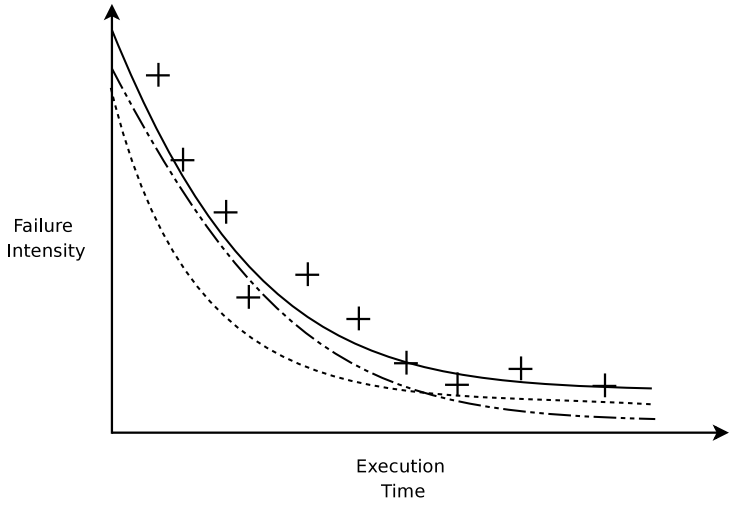


Figure B.2: Different choices for the parameters result in different curves. The probability of getting the observed data on the curve that we want must give maximum probability of obtaining the data.

$x_1 \ldots x_n$ at times $t_1 \ldots t_n$ then from the exponential distribution, $\mathrm{L}(\theta)$ becomes

$$\theta e^{-\theta t_1} \theta e^{-\theta t_2} \ldots \theta e^{-\theta t_n} = (\theta)^n e^{-\theta \Sigma t_i}$$

An estimate for the parameter is then value of θ making

$$\frac{\mathrm{d}(\theta)^{\mathrm{n}} \mathrm{e}^{-\theta \Sigma \mathrm{t_i}}}{\mathrm{d}\theta} = 0 \tag{B.3}$$

which gives a maximum. In general, when the exponential function e is involved take the natural log of $\mathrm{L}(\theta)$ and take the derivative. Doing this gives the same value for θ as the derivative in B.4. In the case of $\mathrm{L}(\theta)$ we get

$$\frac{\mathrm{d} \ln(\theta)^{\mathrm{n}} \mathrm{e}^{-\theta \Sigma \mathrm{t_i}}}{\mathrm{d}\theta} = \frac{n}{\theta} - \Sigma t_i = 0 \tag{B.4}$$

and we can easily solve for θ to get $\theta = \dfrac{n}{\Sigma t_i}$.

The situation with the basic execution time model is not so easy because we have two parameters. To simplify the procedure let $\beta = \frac{\lambda_0}{\nu_0}$ and let X be our random variable whose probability density function is $\lambda_0 \mathrm{e}^{-\beta_1 \tau}$. Next, assume that we have been testing for an execution time period of T_F seconds and have experienced a total of M_F failures to this point. If we repeat the idea above and take *partial derivatives*:

$$\frac{\partial \, \mathrm{L}(\lambda_0, \, \beta_1)}{\partial \, \lambda_0 x} = 0 \qquad \frac{\partial \, \mathrm{L}(\lambda_0, \, \beta_1)}{\partial \, \beta_1} = 0 \tag{B.5}$$

we again arrive at our maximum. We again need to take the natural logarithms of both sides to get

$$\frac{\partial \, \ln \mathrm{L}(\lambda_0, \, \beta_1)}{\partial \, \lambda_0 x} = 0 \qquad \frac{\partial \, \ln \mathrm{L}(\lambda_0, \, \beta_1)}{\partial \, \beta_1} = 0. \tag{B.6}$$

The resulting equations that need to be solved often require numerical methods that are outside of the scope of these notes. For completeness, the two estimators are included below.

$$\frac{M_F \, T_F}{\sum_{i=1}^{M_F} t_i + T_F(\nu_0 - i + 1)} + \sum_{i=1}^{M_F} \frac{1}{\nu_0 - i + 1} = 0 \tag{B.7}$$

$$\frac{M_F}{\beta_1} - \frac{M_F\, T_F}{e^{\beta_1 T_F} - 1} - \sum_{i=1}^{M_F} t_i \quad = \quad 0 \tag{B.8}$$

Appendix C

Tutorials

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING
Tutorial 1

NOTE You are expected to prepare for this tutorial by sketching answers to the tasks and questions before attending the tutorial.

What is Testing Really?

> "*It works! Trust me, I've tested the code thoroughly!*"

. . . and with those bold words many software projects have been made bankrupt.

The aim of this tutorial is to start to get an understanding of our major themes; that is, of the factors that make up quality and how testing effects them. While we have not yet looked at testing formally yet you will be asked in this tutorial to test a small program fragment with the purpose of starting to see where the difficulties of testing lay in practice.

As a first step, let's discuss what testing is, and why we would want to perform testing on our software.

Important Terminology

Before beginning the exercises, consider the following four definitions, the first three of which are defined in Chapter 1 of the lecture notes:

Fault: An incorrect step, process, or data definition in a computer program. Faults are the source of failures – a fault in the program triggers a failure under the right circumstances.

Failure: A deviation between the observed behaviour of a program, or a system, from its specification.

Error: An incorrect *internal* state that is the result of some fault. An error may not result in a failure – it is possible that an internal state is incorrect but that it does not affect the output.

Failures and errors are the result of faults – a fault in a program can trigger a failure and/or an error under the right circumstances. In normal language, software faults are usually referred to as "bugs", but the term "bug" is ambiguous and can mean to faults, failures, or errors; as such, as will avoid this term.

The Programs

The following are taken from the exercises in Chapter 1 of *Introduction to Software Testing* by Offutt and Ammann. For each program, the test case below the program results in a failure that is caused by a fault in the program.

```
/**
 * If x == null throw NullPointerException.
 * Else return the index of the last
 * element in x that equals y.
 * If no such element exists, return -1
 */
public int findLast (int[] x, int y)
{
   for (int i = x.length-1; i > 0; i--)
   {
      if (x[i] == y)
      {
         return i;
      }
   }
   return -1;
}

//Test
//Input: x = [2, 3, 5]; y = 2
//Expected output = 0
```

```
/**
 * If x == null throw NullPointerException.
 * Else return the index of the LAST 0 in x.
 * Return -1 if 0 does not occur in x.
 *
 */
public static int lastZero (int[] x)
{
   for (int i = 0; i < x.length; i++)
   {
      if (x[i] == 0)
      {
         return i;
      }
   }
   return -1;
}

//Test
//Input: x = [0, 1, 0]
//Expected output = 2
```

```
/**
 * If x == null throw NullPointerException.
 * Else return the number of positive
 * elements in x.
 */
public int countPositive (int[] x)
{
   int count = 0;
   for (int i = 0; i < x.length; i++)
   {
      if (x[i] >= 0)
      {
         count++;
      }
   }
   return count;
}

//Test
//Input: x = [-4, 2, 0, 2]
//Expected output = 2
```

```
/**
 * If x == null throw NullPointerException.
 * Else return the number of elements in x
 * that are either odd or positive (or both)
 */
public static int oddOrPos(int[] x)
{
   int count = 0;
   for (int i = 0; i < x.length; i++)
   {
      if (x[i]%2 == 1 || x[i] > 0)
      {
         count++;
      }
   }
   return count;
}

//Test
//Input: x = [-3, -2, 0, 1, 4]
//Expected output = 3
```

Your tasks

For each of the programs, perform the following tasks:

1. Specify the input domain and valid input domain of the function.
2. Identify the fault.
3. If possible, identify a test case that does not execute the faulty statement.
4. If possible, identify a test case that executes the faulty statement, but does not result in an failure.
5. If possible identify a test case that forces the program into an error (that is, the first step that a program deviations from its intended behaviour), but does not produce a failure.
6. Fix the fault and verify that the given test now produces the expected output.

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING

Tutorial 2

Introduction

The aim of this tutorial is twofold. First, it aims to give you some practise at deriving test cases from specifications. Second, it aims for you to start exploring the limits of your test cases, and of the specifications.

The Program

Introduction: LWIG (Less Work Is Good) is a new marking program developed for the department of Information Programming.

The LWIG program takes a .in file as an argument and produces a sorted .out file. The program assumes that the .in file contains at least one student record.

Input File: The input file format is as follows. Each line will contain the data for a single student, and will contain several fields. Each field is separated by a colon.

Each line consists of the following fields, in order:

- A student number, which must be a 5 digit, 6 digit or 9 digit number.
- The student's month of birth, which must be a string of 3 alphabetic characters with the first character capitalised and the remaining as lower case. That is, it must be from the set:

$$\{Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec\}$$

- The date of birth, which must be a number from 1–31 and must be the correct number of days for the month; that is, it cannot be 30 February, because February never has 30 days.
- The student's surname, which must be a string of alphabetic characters all in capitals.
- The first letter of the student's first name, which must be a single capitalised alphabetic character.
- The number of lectures that they slept through, which must be an integer between 0 and 24.

If any input row is invalid, the program should print a warning message and continue with the next record. If the program encounters a more serious problem (e.g. unable to open input file), it will print an error message and exit gracefully.

Example 1 As an example suppose we had the following data.

- Student number: 12345
- Month of Birth: May

- Date of Birth: 26
- Surname: CHAN
- First letter of first name: K
- Slept during one or more lectures: 0

The input line for this data would be `12345:May:26:CHAN:K:0`

Output File: The output file format is as follows. Each line will contain the data for a single student, and each line will contain several fields. Each field is separated by a colon.

Each line consists of the following fields, in order:

- Student number which must be a 5 digit, 6 digit, or 9 digit number.
- Assignment 1 mark: (0–10)
- Assignment 2 mark: (0–10)
- Project mark: (0–30)
- Exam mark: (0-50)
- Final grade: [H1, H2A, H2B, H3, P, HCP, N, N+]
- Comment: String of arbitrary length

Example 2 As an example, consider the following output:

- Student number: 12345
- Assignment 1 mark: 10
- Assignment 2 mark: 10
- Project mark: 25
- Exam mark: 38
- Final grade: H1
- Comment: "Excellent."

The output for this record would be `12345:10:10:25:38:H1:''Excellent''`

Tasks

Question 1

What is the input domain for the LWIG program? What are the input conditions for the LWIG program?

Question 2

Derive input test-cases for the program using equivalence partitioning and boundary-value analysis.

Question 3

Of course, the client has not completely specified the program (but, that is to be expected). There is an additional requirement that the records can be sorted by different output fields.

What are the implications of sorting on the various fields and what test cases would you choose to ensure that sorting has been correctly implemented?

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING

Tutorial 3

Introduction

The purpose of this tutorial is for you to gain a deeper understanding of control- and data-flow techniques, and to compare these to input partitioning techniques.

Talking About Sets

There are still questions about how to characterise, or describe, input/output domains and what level of description and how many words are necessary.

You should aim to get used to characterising sets of values using proper set notation - we are after all, dealing with sets. The set notation does not need to be completely formal for this subject, but we will encourage and expect a mathematical discourse as the subjects goes on and in the exam.

To get the feeling for the level of discourse required read the tutorial and assignment answers on the web-page. Here are some examples of what we expect.

Example 1 Start by giving meaning to all of the (mathematical) variables that you intend to use and then use them to describe the input domain (i.e the set of all inputs that you will use for testing).

(i) Let string be the set of alpha-numeric strings, $\mathsf{str} \in \mathsf{string}$ be any individual alpha-numeric string, and length be a function that returns the length of a string. The input domain is then the set string and the input condition is that

$$\text{for all } \mathsf{str} \in \mathsf{string} \bullet 0 \leq \mathsf{length}[\mathsf{str}] \leq 100.$$

Equivalently we could have written the input condition as:

$$\forall\, \mathsf{str} \bullet \mathsf{str} \in \mathsf{string} \implies 0 \leq \mathsf{length}[\mathsf{str}] \leq 100.$$

Both are acceptable.

(ii) Let int be the set of all 32-bit integers and string be the set of all strings of characters. The input domain is then the set of all pairs

$$\mathsf{Input} = \{(\mathsf{x}, \mathsf{y}) \mid \mathsf{x} \in \mathsf{string} \text{ and } \mathsf{y} \in \mathsf{int}\}.$$

Working With the Program

In this tutorial we will focus on the procedure `bubble`, found in Figure C.1. The `main` program serves to show you how we intend to use `bubble`. Of course, `bubble` implements a bubble sort. The source code can be downloaded from the tutorials page of the LMS.

```
#include <stdio.h>
#define SIZE 10

void bubble(int data[SIZE], int size, int (*compare)(int, int));

main()
{
  int data[SIZE] = {11, 4, 8, 22, 15, 7, 8, 19, 20, 1};
  int counter, order;

  int up(int, int);
  int down(int, int);

  printf("Enter 0 to sort in ascending order and 1 to sort in descending order: ");

  scanf("%d", &order);

  if (order == 0) {
    bubble(data, SIZE, up);
  }
  else {
    bubble(data, SIZE, down);
  }

  for (counter = 0; counter < SIZE; counter++) {
    printf("%4d", data[counter]);
  }

  printf("\n");
}

void bubble(int *data, int size, int (*compare)(int, int))
{
  int pass, count;
  void swap(int *, int *);

  for (pass = 0; pass < SIZE - 1; pass++) {
    for (count = 0; count < SIZE - 1; count++) {
      if ((*compare)(data[count], data[count + 1])) {
        swap(&data[count], &data[count + 1]);
      }
    }
  }
}

void swap(int *A, int *B)
{
  int temp;

  temp = *A;
  *A = *B;
  *B = temp;
}
```

Figure C.1: An implementation of bubble sort

Make special note of the third parameter to `bubble`, which is a function parameter. This parameter allows us to create a flexible sorting algorithm, which can be used to sort in either ascending or descending order. Both

functions take two integers and return an integer (which is intended to be a boolean value). For this tutorial consider just

```
int up(int A, int B) {
  return A < B;
}
```

and

```
int down(int A, int B) {
  return B < A;
}
```

as the possible comparison functions. Note that the functions `up` and `down` appear in the `main` function.

Another interesting aspect of both `bubble` and `main` below is that they declare *nested functions*. The functions `up` and `down` are declared within `main` and so can refer to any of the variables in `main` prior to their declaration, for example, they can refer to the variable `count`.

Your Tasks

Task 1

First, make sure you understand the program (expected to be done **before** the tutorial).

Task 2

Determine the input domains and output domains for the functions `bubble`. What is the set of possible functions that can be passed into the `bubble` function? Is this what you expected?

Next, what are the input conditions, and (perhaps a little more challenging) what are the output conditions?

Task 3

Create a control-flow graph for the `bubble` function. Ideally, should you include the functions `up` and `down` in your control flow graph?

Question 4

Do the branches in your control flow graph partition the input domain? Do the branches in your control flow graph partition the output domain?

Task 5

Perform a static data-flow analysis on `bubble` for the variables `data` and `size`.

Task 6

Design a set of black box test cases using whatever techniques you feel appropriate.

Question 7

What extra information does your data-flow analysis give you that your black-box test cases do not?

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING

Tutorial 4

Introduction

The aim of this tutorial is for you to familiarise yourself with the various coverage criteria and analysis of the program for the various coverage criteria. When you get back to your revision you should try comparing the test cases that you derive for a program using different techniques.

The different type of program that we encounter this week is a numerical program. One of the challenges of numerical programs is that we can never be certain that we will get an *exact* answer to our computation. Instead what we typically require is an answer to within some *error* value[1]. Numerical programs are tricky to debug, because they are often used to *find* the answer to some problem in the first place. For example, solving some integration or differentiation problems is too hard to do by hand and so we use a *numerical* method to approximate the answer.

Working With the Program

The program implements the standard bisection method for root finding. The root-finding problem is expressed as follows:

> We are given a function $f(x)$ taking a real number and returning a real number[2]. The function is negative at some point x_0 and positive at some point x_1. Find the value x for which $f(x) = 0$ on an interval $[Lower, Upper]$. The point x is a root of f on the interval $[Lower, Upper]$.

As an example, consider the natural logarithm function, ln. The graph in Figure C.2 shows the values of $ln(x)$ for various values of x. We can see that the value of $ln(x)$ is equal to 0 when $x = 1$. The bisection algorithm finds this value of x.

The idea behind the algorithm for finding roots is to look at the interval $[Lower, Upper]$ and bisect it (hence the name of the algorithm) and find the midpoint of the interval x_r. If we know that $f(Lower)$ is negative, and $f(Upper)$ is positive then there must be root in the interval, provided that the function is continuous. If the value of f at x_r is positive then the root must be in the interval $[Lower, x_r]$. If the value of f at x_r is negative then the root must be in the interval $[x_r, Upper]$. The algorithm should converge to the root because the length of the interval is getting smaller every time (in fact the length of the interval is halved every time). Does this sound familiar?

[1] Recall from the lecture notes that an error is the difference between a computed value and the exact value.
[2] That is a function $f : \mathbb{R} \to \mathbb{R}$.

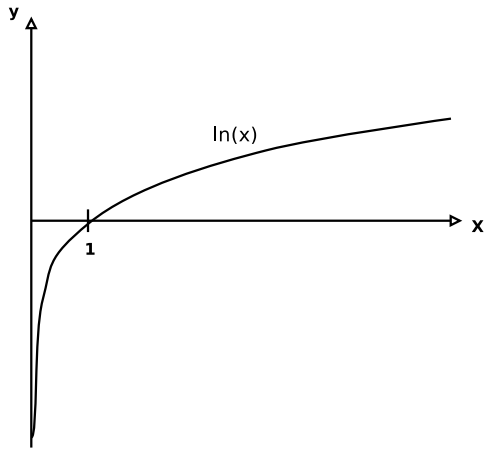


Figure C.2: Root finding problems.

Your Tasks

Question 1

What is the input domain for the `Bisection` program below?

Question 2

Draw the control-flow graph for the `Bisection` function. You may break the function up into basic blocks to simplify your CFG.

Recall that a *basic block* is a continuous sequence of statements where control flows from one statement to the next, a single point of entry, a single point of exit and no branches or loops.

Question 3

Suppose that we concentrated on the (nice and linear) function $f(x) = x - 2$. Derive a set of test cases that achieve:

(i) Statement coverage; and

(ii) Condition coverage.

Note, that you will have to determine what it means for the `Bisection` function to return the *correct* or *expected* output first.

The Program

```
#include <stdio.h>
#include <math.h>

#define MAX_INT 65535

double Bisection(double Lower,         /* Lower bound of the interval. */
                 double Upper,         /* Upper bound of the interval. */
                 double error,         /* Allowed error. */
                 int iMax,             /* Bound on the number of iterations. */
                 double (*f)(double) ) /* The function. */
{
  double Sign = 0.0;   /* Test for the sign of the midpoint xr. */
  double ea = MAX_INT; /* Calculated error value. */
  double xrold = 0.0;  /* Previous estimate.  */
  double xr = 0.0;     /* Current x estimate for the root. */
  double fr = 0.0;     /* Current value of f. */
  double fl = 0.0;     /* Value of f at the lower end of the interval. */
  int iteration = 0;   /* For keeping track of the number of iterations. */

  fl = (*f)(Lower);
  while (ea > error && iteration < iMax) {

    /* Start by memorising the old estimate in xrold and then calculate
       the new estimate and store in fr */

    xrold = xr;
    xr = (Lower + Upper) / 2;
    fr = (*f)(xr);

    iteration++;

    /* Estimate the percentage error and store in ea. */

    if (xr != 0) {
      ea = fabs((xr - xrold)/xr) * 100;
    }

    /* To know whether fr has the same sign as f(Lower) or f(Upper) is easy:
       we know that f(Lower) is negative and we know that f(Upper) is positive.
       Multiple fr by f(Lower) and if the result is positive then fr must be
       negative. If the result is negative then fr must be positive. */

    Sign = (*f)(Lower) * fr;

    if (Sign < 0)
      Upper = xr;
    else if (Sign > 0)
      Lower = xr;
    else
      ea = 0;

    printf("iteration %d = (%f, %f, %f, %f, %f)\n", iteration,
            Lower, Upper, xr, ea, Sign);

  } /* end while */

  return xr;
```

```
}

int main()
{
  double f(double);

  double fx = Bisection(-1.0, 7.0, 1, 10, &f);
  printf("value = %f\n", fx);
}

double f(double x)
{
  return x - 2;
}
```

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING

Tutorial 5

Introduction

Encapsulation is an abstraction mechanism that aids in programming, but that adds complexity to testing. We often need to break the information hiding utilised by classes in order to examine the class state for the testing purposes.

The aim of this tutorial is for you to explore some of the issues in object oriented testing through the simple Graph given below. The graph uses an adjacency matrix that records which vertices are "*adjacent*" in the graph.

Task 1

Consider the `addEdge` and `deleteEdge` methods in the `Graph` class below. Derive test cases for path coverage and condition coverage for these two methods. **Note** that it may be necessary to examine the state variables in your test cases. Sketch how you would achieve this.

Task 2

Draw a finite state automaton for `Graph` class. **Note** that it is not always possible to add an edge and it is not always possible to delete a vertex. You will need to consider the states and the guards on transitions to ensure all of the conditions.

Task 3

Derive a set of test cases to test every transition in your graph.

Question 4

Using 5 different mutant operators from Chapter 6 of the lecture notes, derive 5 mutants for the `addEdge` operation. Do the test cases derived from your finite state automaton kill all of the mutants? If not, are the live mutants equivalent, or are the test cases not yet adequate?

The Graph Abstract Data Type

The Graph Interface

```
class Graph
{
    public Graph(int n);
      // The constructor for graphs. It initialises
      // the graph state and sets up the graph
      // representation for graphs of n or less nodes.

    public void addVertex(int v);
      // Add a vertex to the vertex graph.

    public void addEdge(int m, int n)
      // Add an edge to the graph. Edges are specified by pairs
      // of vertices. To add the edge correctly it is necessary
      // that m and n have already been added to graph as vertices.

    public void deleteVertex(int v);
      // We can only delete a node if it is not part of some edge
      // in the graph and it exists as an actual vertex in the
      // graph.

    public void deleteEdge(int m, int n);
      // We can only delete an edge if the two specified
      // vertices, m and n, are in the graph.
}
```

The Graph State and Hidden Functions

```
//------------ Private Attributes ----------------------------
//    The representation of a graph consists of an array
//    of vertices that map nodes (just integers) to array
//    indexes. The adjacency matrix _matrix sets matrix[i][j]
//    to true if there is an edge between _vertices[i] and
//    _vertices[j].
//
//    The operations must maintain the following invariant:
//        _vertices[i] is defined iff i < _allocated
//        _allocated <= _order

private int     _order;      // The number of vertices allowed
private int     _allocated;  // The next free space in the vertex array
private int     _vertices[]; // A list of the actual vertices
private boolean _matrix[][]; // The adjacency matrix

//------------- Private Methods --------------------------------

private static boolean[][] _allocate(int n)
{
  return new boolean[n][n];
}

private static int[] _vertices(int n)
{
  return new int[n];
}

private int _lookup(int m)
{
  int index = 0;
  while (index < _allocated && !(_vertices[index] == m))
    index = index + 1;

  if (index == _allocated) {
    return _order + 1;
  }
  else {
    return index;
  }
}
```

Adding and Deleting Edges

```
public void addEdge(int m, int n)
{
  // Add an edge to the graph. Edges are specified by pairs
  // of vertices. To add the edge correctly it is necessary
  // that m and n have already been added to graph as vertices.

  int mIndex = _lookup(m);
  int nIndex = _lookup(n);

  if (mIndex < _order && nIndex < _order)
  {
    _matrix[mIndex][nIndex] = true;
    _matrix[nIndex][mIndex] = true;
  }
}

public void deleteEdge(int m, int n)
{
  // We can only delete an edge if the two specified
  // vertices are in the graph.

  int mIndex = _lookup(m);
  int nIndex = _lookup(n);

  if (mIndex < _order && nIndex < _order) {
    _matrix[mIndex][nIndex] = false;
  }
}
```

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING

Tutorial 6

Introduction

The aim of this tutorial is for you to familiarise yourself with the test oracles and random testing. This tutorial will help to gain an understanding of how to specify a test oracle to decide whether an arbitrary test case is correct, incorrect or undecidable, as well as how to randomly generate test cases that fit an operational profile.

Your Tasks

Repeat these tasks for both programs.

Task 1

Standard analysis: Determine the input/output domains and the input/output conditions.

Task 2

Use the specification, domains and conditions to determine if an arbitrary test input is correct, incorrect or otherwise undecidable.

Task 3

Determine an automated means for generating random test cases by selection points from the input set.

Working With the Programs

We shall start this tutorial by using a rather simple example to illustrate the concepts, techniques and issues faced when working with oracles. This is followed by applying these techniques to the root finding program from tutorial 4 to gain some practice with practical examples.

First Program

This *toLower* is an application which takes a string on standard input and puts a corresponding string on standard output, with all upper case letters changed to the matching lower case letters (i.e. 'A' to 'a', 'B' to 'b', etc.).

Strings may consist of all ASCII characters between 32 and 126 inclusive; characters outside this range are control characters and will cause an error message.

toLower Program

```
#include <stdio.h>
#include <string.h>

#define MAX_STRING 80

int main(int argc, char* argv[])
{
  char string[MAX_STRING];
  int i;
  while (fgets(string, MAX_STRING, stdin)) {
    for (i = 0; i < strlen(string) - 1; i++) {
      if (string[i] < 32 || string[i] == 127) {
        printf("Illegal character found.\n");
        return 1;
      }
      /* The upper case letters have ASCII codes
       * between 65 and 90 (90 to 65 = 26 letters)*/
      if (string[i] >= 65 && string[i] <= 90) {
        string[i] = string[i] + 'a' - 'A';
      }
    }
    fputs(string, stdout);
  }
  return 0;
}
```

Second Program

The idea behind the algorithm for finding roots is to look at the interval $[Lower, Upper]$ and bisect it (hence the name of the algorithm) and find the midpoint of the interval x_r. If we know that $Lower$ is negative, and $Upper$ is positive then there must be root in the interval, provided that the function is continuous. If the value of f at x_r is positive then the root must be in the interval $[Lower, x_r]$. If the value of f at x_r is negative then the root must be in the interval $[x_r, Upper]$. The algorithm should converge to the root because the length of the interval is getting smaller every time (in fact the length of the interval is halved every time).

Root Finding Program: A Simple Interval Bisection Method for Finding Roots

```
#include <stdio.h>
#include <math.h>

#define MAX_INT 65535

double Bisection(double Lower,         /* Lower bound of the interval. */
                 double Upper,         /* Upper bound of the interval. */
                 double error,         /* Allowed error. */
                 int iMax,             /* Bound on the number of iterations. */
                 double (*f)(double) ) /* The function. */
{
  double Sign = 0.0;   /* Test for the sign of the midpoint xr. */
```

```
  double ea = MAX_INT; /* Calculated error value. */
  double xrold = 0.0;  /* Previous estimate.  */
  double xr = 0.0;     /* Current x estimate for the root. */
  double fr = 0.0;     /* Current value of f. */
  double fl = 0.0;     /* Value of f at the lower end of the interval. */
  int iteration = 0;   /* For keeping track of the number of iterations. */

  fl = (*f)(Lower);
  while (ea > error && iteration < iMax) {

    /* Start by memorising the old estimate in xrold and then calculate
       the new estimate and store in fr */
    xrold = xr;
    xr = (Lower + Upper) / 2;
    fr = (*f)(xr);

    iteration++;

    /* Estimate the percentage error and store in ea. */
    if (xr != 0) {
      ea = fabs((xr - xrold)/xr) * 100;
    }

    /* To know whether fr has the same sign as Lower or Upper is easy:
       we know that Lower is negative and we know that Upper is positive.
       Multiple fr by Lower and if the result is positive then fr must be
       negative! If the result is negative then fr must be positive! */

    Sign = (*f)(Lower) * fr;

    if (Sign < 0)
      Upper = xr;
    else if (Sign > 0)
      Lower = xr;
    else
      ea = 0;

  } /* end while */

  return xr;
}
```

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING

Tutorial 7

NOTE You are expected to prepare for this tutorial by sketching answers to the tasks and questions before attending the tutorial.

Introduction

The aim of this tutorial is to get you thinking about software security and vulnerabilities, and the applicability of different kinds of security testing.

As a first step, think about what security testing is, and why we would want to perform security testing on our software.

The Bitmap File Format

BMP is an historical image file format that we will use in this tutorial. We will consider a simple class of BMP files whose format is as follows. (Specifically, we consider here BMP files with no compression, and in which each pixel is 32-bits wide in order to avoid issues of padding; see `http://www.fastgraph.com/help/bmp_header_format.html` and `https://en.wikipedia.org/wiki/BMP_file_format` for more details.)

Offset	Size (in bytes)	Description
0	1	first byte of signature, must be 0x42 (the ASCII character 'B')
1	1	second byte of signature, must be 0x4D (the ASCII character 'M')
2	4	size of the BMP file in bytes (unreliable, ignored)
6	2	Must be zero
8	2	Must be zero
10	4	Must be the value 54 (i.e. 0x00000036)
14	4	Must be the value 40 (i.e. 0x00000028)
18	4	*Width* (image width in pixels, as signed integer)
22	4	*Height* (image height in pixels, as signed integer)
26	2	Must be one
28	2	Number of bits per pixel (must be 32)
30	4	Compression type (must be 0 = no compression)
34	4	Size of image data in bytes (must be 4**Width***Height*)
38	4	unreliable (ignored)
42	4	unreliable (ignored)
46	4	Must be zero
50	4	Must be zero
54	4**Width***Height*	Pixel data, laid out in rows

The first byte (offset 0) of a valid BMP file is the character 'B'; the second byte (offset 1) is the character 'M'. The 3rd to 6th bytes (offsets 2 to 5 inclusive) indicate the total length of the BMP file but are unreliable in practice and so let us assume that they are ignored by all BMP parsing code. The 7th and 8th bytes (offsets 6 and 7) are interpreted as a 2-byte integer that must be zero, i.e. each of these bytes must be zero. The same is true for the 9th and 10th bytes (offsets 8 and 9), and so on.

Your Tasks

Question 1

Imagine you are choosing a value for each of the fields in the table above *in order*, i.e. you first choose a value for the first byte of the file, then choose a value for the second byte of the file, then for following 4-bytes, and so on. For each field, identify the total number of valid values there are to choose from, assuming you have already chosen values for all fields that have come before.

Question 2

The BMP header (i.e. everything excluding the pixel data) as described above has a fixed length of 54 bytes. Using the answer from the previous question or otherwise, what is the probability that a (uniformly) randomly generated string of 54 bytes is a valid BMP header?

Question 3

Suppose you have a valid 54-byte header and you mutate an arbitrary (uniformly randomly chosen) byte in the header to a new value (different from its original value). What is the probability of producing a valid header?

Question 4

Imagine you had to write a fuzzer to fuzz some BMP processing code that processed BMP files of the format described above. If you had to choose between generating completely random inputs vs. using random mutation on existing BMP files, which strategy would you choose?

Question 5

Suppose you are fuzzing some C code that reads BMP files and you observe that for some large set of inputs generated by your fuzzer that the code crashes some number N of times. What would happen if you ran the code on the same inputs but now under a memory debugger? For example, imagine you recompile the BMP parsing code using the ASan (Address Sanitizer) tool demonstrated in lectures and ran it again on the same fuzzer-generated inputs as before. Would you expect the number of crashes observed now to exceed N, be the same, or be less than N? Explain why.

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING

Tutorial 8

Introduction

The aim of this tutorial is to give you more experience with symbolic execution and dynamic symbolic execution, and how they can be used to verify software.

(Dynamic) Symbolic Execution

Recall from the lecture that the aim of symbolic execution is to, in effect, execute *multiple* inputs at the same time – possibly an infinite number. It does this by uses symbolic values to represent inputs, instead of single concrete values.

A complete symbolic execution of a program produces a *symbolic execution tree*. Each path through the tree represents a single path through the program, including all inputs on that path. Each node along a path represents the symbolic values of variables at that point in the execution.

Dynamic symbolic execution, on the other hand, follows the path taken by a concrete test input, and collects the relevant constraints along the way. It uses these constraints to generate tests that take new paths in the program.

Tasks and Questions

Question 1

Figure C.3 shows an implementation of a program that returns the minimum of two integers. Draw the complete symbolic execution tree for this program. In your execution tree, use the variables `X_0` and `Y_0` to represent the initial symbolic values of `x` and `y` respectively.

At the return statement at line 10, assume the existence of a symbolic variable `RET`, to which the return value is assigned when a return statement is executed.

Question 2

Does the program in Figure C.3 ensure the precondition that the number returned is always the minimum; i.e. that `RET` is the minimum number of `X_0` and `Y_0`?

Question 3

Consider the program in Figure C.4 for counting the number of negative integers in an array. Assume that we run a *dynamic symbolic execution* (DSE) tool on this program, with the initial test being `x = [0,0]`.

```
 1.  int min(int x, int y)
 2.  {
 3.      int minimum = 0;
 4.      if (x > y) {
 5.          minimum = y;
 6.      }
 7.      else if (y > x) {
 8.          minimum = x;
 9.      }
10.      return minimum;
11.   }
```

Figure C.3: An implementation of the Min function

```
 1.  public int countPositive (int[] x)
 2.  {
 3.      int count = 0;
 4.      for (int i = 0; i < x.length; i++) {
 5.          if (x[i] < 0) {
 6.              count++;
 7.          }
 8.       }
 9.       return count;
10.  }
```

Figure C.4: The `countPositive` function

Specify the path constraint that this will generate.

Then, using the path constraint from this, flip one of the constraints corresponding the branch at line 5, specify a test that satisfies this, and its path constraint.

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING

Tutorial 9

Introduction

Reliability block diagrams are a simple and powerful method for making reliability assessments for systems. There are a couple of things to be wary about however:

- We need to be careful that all failure probabilities are for the same time period T;
- We should not confuse failure logic with reliability logic.

The aim of this tutorial is to give you practice at taking system descriptions, determining where failures can occur and creating parallel-serial reliability block models for those systems. Along the way we also get a brief look at Triple-mode Redundant systems – an instance of NMR, or N-mode Redundant, systems.

Understanding Reliability Blocks

Reliability block diagrams usually need to be created to model the reliability of a system. Typically we are given a system diagram that we need to analyse for reliability. To do this we create the reliability block diagram for the system and then perform the analysis on it. There are several steps involved in creating reliability block diagrams from system architectures:

Step 1 Determine what constitutes a *system* failure. This in turn determines which component failures cause a system to fail.

Step 2 Determine which components need to fail in order to cause a system failure. Some guidelines for constructing reliability block models is to consider how messages, signals or data flow through the system.

Step 3 Calculate system reliability from your reliability block diagram using either reliability logic or failure logic.

Some guidelines for creating reliability block models.

- Try to ensure that each block in the reliability block model captures one function of the system.
- Try to ensure that you capture the parallel/serial connections from the system accurately in your reliability block model.
- There may be more than one mode for the system - you will need to create a reliability block diagram for each mode of the system.

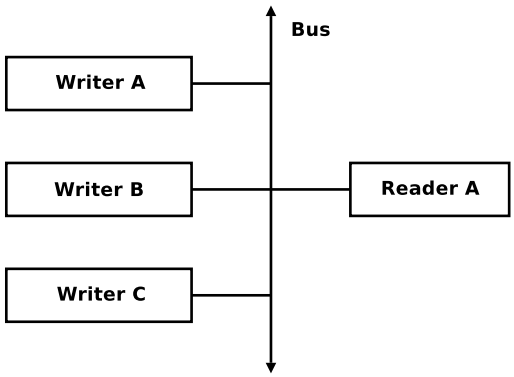


Figure C.5: Readers and writers communicating over a shared bus.

Working with the Sample System

Question 1

Consider the readers and writers system in Figure C.5. Three redundant writers Writer A, Writer B, and Writer C each read the same source to obtain a value. They then send that value via a Bus to Reader A. Each writer acts by placing a message on the bus when it has a message to send (assume that there is no contention between readers and writers for the bus). The reader then takes the message and processes it.

The system fails if no value is processed within a given time period T.

(i) List the ways in which the system can fail.

(ii) Next create a reliability block model for the writers, bus and reader.

(iii) After some measurement, it is estimated that the components have the following failure probabilities.

Component	Failure Probability in time T
Writer A	0.01
Writer B	0.05
Writer C	0.01
Bus	0.02
Reader	0.01

Calculate the *failure probability* and the *reliability* for the system with respect to time T.

Question 2

Consider the system of servers, routers and databases shown in Figure C.6.

The system consists of servers, routers and databases.

Each of the servers performs the same function. It makes external connections and serves pages. The server selects a database, retrieves the pages and other data and sends them back along the connection.

As long as one server is still working connections can be made at the cost of some degradation in performance.

The routers perform the task of connecting a server with a databases. Routers also handle faults in the system. If they detect a faulty databases machine or a faulty server they can then route the other traffic around these.

The three databases are redundant data stores and each database contains an exact duplicate of the data. Only one of the databases needs to be functioning at any one time.

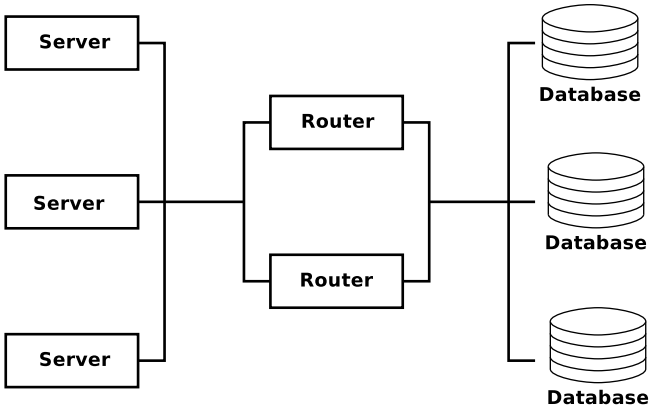


Figure C.6: A system of servers and databases communicating through duplicated routers.

(i) Define what it means for the system to fail and then determine which sets of components need to fail for the system to fail.

(ii) Create a reliability block model for the system.

(iii) Assume that the failure rate over 1 day of the three servers is 0.01, 0.02, and 0.03 respectively, the failure rate of each router is 0.02, and the failure rate of each database is 0.02. What is the reliability of the system over 2 days?

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING

Tutorial 10

Introduction

The aim of this tutorial is to give you a better understanding of Markov models, and how they can be used to help estimate system reliability.

Recall that a Markov model is a finite state machine in which the transitions have a probability attached to them representing the probability of that particular transition being taken. By mapping a Markov model to a matrix, calculating the probability of being in particular states can be achieved using a straightforward matrix multiplication (see Tutorial 1 for a matrix multiplication algorithm).

Tasks and Questions

Question 1

Consider a simple sensing device. The manufacturer gives you the following failure data for the device based on their measurements.

State	**Per-hour Failure Probability**	**Per-hour Repair Probability**
Functional $\rightarrow$ Degraded	10^{-5}	10^{-3}
Degraded $\rightarrow$ Non-Operational	10^{-2}	0
Functional $\rightarrow$ Non-Operational	10^{-6}	0

Table C.1: States and failure rates for the simple sensing device.

Table C.1 shows the probabilities for making transition between functional, degraded, and non-operational states (in the **Per-hour Failure Probability** column) in one hour of operation. It also gives the probability of repair (in the **Per-hour Repair Probability** column) in an hour; that is, the probability that the system transitions back to a functional state.

Draw the Markov model (in the form of a state transition diagram) with *all* of the transition probabilities clearly labelled. Determine the transition matrix for this model.

Question 2

Consider a simple system involving a node linked to a noisy channel. After some measurement the data in the following table is obtained. Assume that failures and repairs are independent.

(i) Determine the Markov states for the system.

Device	**Per-hour failure probability**	**Per-hour repair probability**
Node	10^{-4}	10^{-3}
Channel	10^{-3}	10^{-2}

Table C.2: Failure data for a simple noisy channel.

(ii) Draw the state transition diagram showing all of the transition probabilities.

(iii) Calculate the 1-step transition matrix for your model.

THE UNIVERSITY OF MELBOURNE
SWEN90006: SOFTWARE & SECURITY TESTING

Tutorial 11

Introduction

The aim of this tutorial is to start you thinking about reliability measurement and in manipulating and interpreting the different reliability formulae. Before beginning, there are a couple of things to note about reliability measurement, or any form of software measurement, for that matter.

In reliability measurement, we need to consider the following points:

1. **Deciding What "Failure" Means for an Application** — In reliability measurement you will need to decide what is means for a module to *fail*. To understand what constitutes a failure you will typically need to understand the module specifications in some detail.
2. **Deciding What to Measure** — Remember that we do not test every input with equal probability. Reliability estimates are based on an operational profile of the the system. If the estimated operational profile is not close to the actual operational profile then reliability estimates will be less accurate.
3. **Detecting Failures** — Specifying invariants for classes and specifying conditions for failures to occur are part of designing a test oracle for finding failures. Of course, the problem with writing test oracles is that we may not be able to detect every single failure. Thus, we may get a lower bound on the actual number of failures.
4. **Recording Failure Times** — Recording the failure times is also problematic as the time at which failures are experienced and the times between failures can NOT be measured accurately. However, the actual error in time measurement is often small compared to execution time scales. For example, an error of 10 milliseconds is negligible if the time between failures is measured in minutes.

Given that reliability measurement is inexact we need to understand: (1) *exactly* what we can say about reliability measurement; (2) how to interpret the data; and (3) how to make the best use from the information that we have.

Tasks and Questions

Question 1

Using the regression equations and the data from the appendix, calculate the following:

(i) the initial failure intensity λ_0; and

(ii) the total number of failures ν_0;

Question 2

Using the regression equations and the data from the appendix, calculate the following:

(i) the expected number of failures after 10 000 CPU seconds; and

(ii) the expected failure intensity after 10 000 CPU seconds.

Question 3

Consider now two possible target failure intensities:

(i) one failure every 12 hours; and

(ii) one failure every 24 hours.

How many more failures would need to experienced before *each* of these two target failure intensities is reached?

Question 4

How much more execution time for testing would be required for *each* of these two target failure intensities to be reached?

Question 5

Consider the two failure intensity against failures experienced graphs in Figures C.7 and C.8 respectively. Suppose that the equation for the regression line in Figure C.7 is:

$$\lambda_A(\mu_A) = \lambda_{A_0}\left(1 - \frac{\mu_A}{\nu_{A_0}}\right) \tag{C.1}$$

and the equation for the regression line in Figure C.8 is:

$$\lambda_B(\mu_B) = \lambda_{B_0}\left(1 - \frac{\mu_B}{\nu_{B_0}}\right) \tag{C.2}$$

Notice that the slope of the line $\lambda_B(\mu_B)$ is steeper than the slope of $\lambda_A(\mu_A)$; that is,

$$-\frac{\lambda_{A_0}}{\nu_{A_0}} \geq -\frac{\lambda_{B_0}}{\nu_{B_0}}. \tag{C.3}$$

Discuss how each of the model parameters obtained from the lines in Figure C.7 and C.8 affect:

(i) The expected number of failures $\mu(t)$; and

(ii) The failure intensity $\lambda(t)$.

Appendix

The data in Table C.3 records the set of failure times for an actual system, which we call T_1. That is, the first failure occurred after 3 CPU seconds, the second failure after a further 30 seconds (33 seconds), the third after a further 111 seconds (146), etc. The test phase ended after 6600 seconds of execution time. Table C.4 summarises the data from Table C.3.

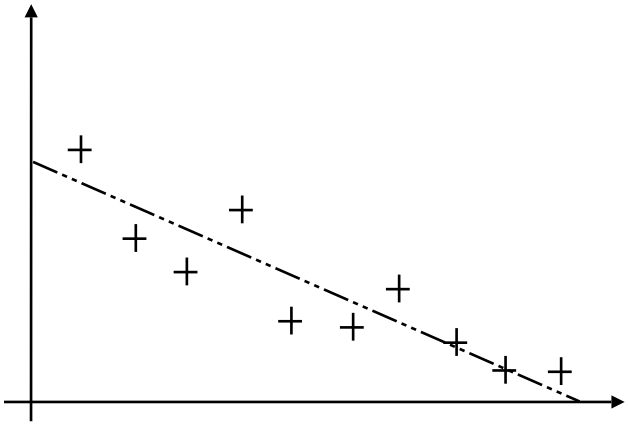


Figure C.7: Failure intensity vs failures experienced - Graph A

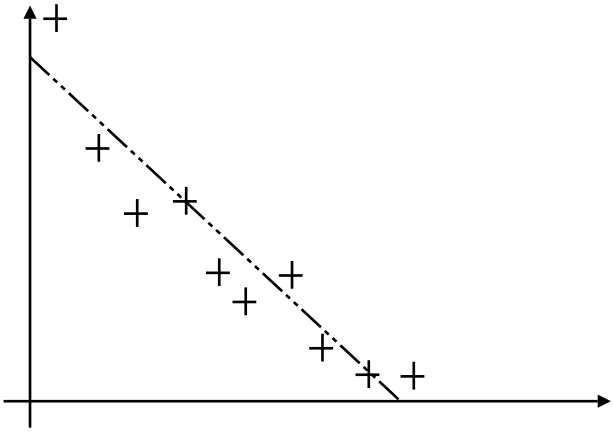


Figure C.8: Failure intensity vs failures experienced - Graph B

3	709	2311	5085
33	759	2366	5089
146	836	2608	5097
227	860	2676	5324
342	968	3098	5389
351	1056	3278	5565
353	1726	3288	5623
444	1846	4434	6080
556	1872	5034	6380
571	1986	5049	6477

Table C.3: Time of Failure for System T_1 (CPU seconds)

Failures	Time (CPU Sec.)	Failure Intensity (Failures/Second)
5	342	0.01462
10	571	0.02183
15	986	0.01205
20	1986	0.005
25	3098	0.004496
30	5049	0.002563
35	5389	0.014706
40	6477	0.004596

Table C.4: Failure Time Data for System T_1

Regression Equation

The regression equation (mean no. of failures experienced against Failure intensity) is

$$\lambda = -0.00032\mu + 0.017203$$