Software Testing

MCA Third Year
Paper No. X

School of Distance Education
Bharathiar University, Coimbatore - 641 046

Author: Isha Chadha

Produced and Printed
by
EXCEL BOOKS PRIVATE LIMITED
A-45, Naraina, Phase-I,
New Delhi-110 028
for
SCHOOL OF DISTANCE EDUCATION
Bharathiar University
Coimbatore - 641 046

CONTENTS

Page No.

UNIT I

Lesson 1 Introduction to Software Testing 7

Lesson 2 The Taxonomy of Bugs 17

UNIT II

Lesson 3 Software Testing Techniques 35

Lesson 4 Flowgraphs and Path Testing 45

UNIT III

Lesson 5 Transaction Flow Testing 67

Lesson 6 Data Flow Testing 75

Lesson 7 Syntax Testing 88

UNIT IV

Lesson 8 Logic-based Testing 101

Lesson 9 States, State Graphs and Transition Testing 122

UNIT V

Lesson 10 Testing Specialized Environments, Architecture and Applications 137

Lesson 11 Testing Tactics and Debugging 158

Model Question Paper 179

SOFTWARE TESTING

SYLLABUS

UNIT I

Purpose of Software testing - Some Dichotomies - a model for testing - Playing pool and consulting oracles - Is complete testing possible - The Consequence of bugs - Taxonomy of Bugs.

UNIT II

Software testing Fundamentals - Test case Design - Introduction of Black Box Testing and White Box testing - Flow Graphs and Path testing - Path testing Basics - Predicates, Path Predicates and Achievable Paths - Path Sensitizing - Path Instrumentation - Implementation and Application of Path Testing.

UNIT III

Transaction Flow testing - Transaction Flows - techniques - Implementation Comments - Data Flow Testing - Basics - Strategies - Applications, Tools and effectiveness - Syntax Testing - Why, What, How - Grammar for formats - Implementation - Tips.

UNIT IV

Logic Based Testing - Motivational Overview - Decision tables - Path Expressions - KV Charts - Specifications - States, State Graphs and transition Testing - State Graphs - Good & bad states - state testing Metrics and Complexity.

UNIT V

Testing GUIs - Testing Client - Server Architecture - Testing for Real-time System - A Strategic Approach to Software testing - issues - unit testing - Integration Testing - Validation testing - System testing - The art of Debugging.

UNIT I

LESSON

1

INTRODUCTION TO SOFTWARE TESTING

CONTENTS

1.0 Aims and Objectives
1.1 Introduction
1.2 Purpose of Software Testing
1.3 Some Dichotomies
  1.3.1 Testing versus Debugging
  1.3.2 Function versus Structure
  1.3.3 The Designer versus the Tester
  1.3.4 Modularity versus Efficiency
  1.3.5 Small versus Large
  1.3.6 The Builder versus the Buyer
1.4 A Model for Testing
  1.4.1 The Environment
  1.4.2 The Program
  1.4.3 Bugs
  1.4.4 Tests
  1.4.5 Testing Levels
  1.4.6 The Role of Models
1.5 Playing Pools and Consulting Oracles
  1.5.1 Playing Pool
  1.5.2 Oracles
1.6 Is Complete Testing Possible?
1.7 Let us Sum up
1.8 Lesson End Activities
1.9 Keywords
1.10 Questions for Discussion
1.11 Suggested Readings

1.0 AIMS AND OBJECTIVES

After studying this lesson, you would be able to understand:

- The purpose of testing
- Roles of models
- Playing pools and consulting oracles
- A model for testing

1.1 INTRODUCTION

Software testing is a process, or a series of process, designed to ensure that the computer code does what it is designed for and does not do something that is not intended. Software must be predictable and consistent, offering no surprises to its users.

Testing involves at least half of the time and efforts spent to build working software. And the efforts put into testing are considered waste if the tests conducted on the code do not reveal all the errors.

Testing ensures presence of errors, not the absence of errors. Even after a code has been tested thoroughly, it can report errors. So, even the best written code can have errors. The only way to minimize it is to devise best possible test design and carry out testing which covers maximum possible functionality.

1.2 PURPOSE OF SOFTWARE TESTING

While testing a program, it is required to add value to it. This would mean increasing the quality or reliability of the program i.e. finding errors and removing them.

Thus, one must not test a program to show that it works but with an aim to reveal errors and that too as much as possible.

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

Thus, the objectives of testing can be listed as below:

- Executing a program in order to find errors.
- A good test case is the one that has a high probability of finding an undiscovered error.
- A successful test is the one that reports an as–yet undiscovered error.

The objective while designing tests is to come up with the test cases that systematically uncover different classes of errors in minimum possible time and effort.

1.3 SOME DICHOTOMIES

1.3.1 Testing versus Debugging

Often, testing and debugging are grouped in the same category and are confused with one another. However, the purpose of testing is to show the presence of errors in a program and the purpose of debugging is to find errors or misconception that led to the failure of the program and to design and implement the changes that can correct these errors. Debugging normally follows testing.

- Testing begins with known conditions, uses pre-defined procedures and has predictable outcomes. Debugging starts from possibly unknown initial conditions and the end cannot be predicted, except statistically.
- Testing can be planned, designed and scheduled. However, the procedures and duration of debugging cannot be constrained.
- Testing detects the errors and debugging is a deductive process.
- Testing proves a programmer’s failure and debugging a programmers’ vindication.
- Testing is subset of verification and validation.

- Testing can be done by an outsider, but debugging must be done by an insider.
- Test execution and design can be automated. Automated debugging is still a vision.

1.3.2 Function versus Structure

Tests can be designed either functionally or structurally. In the functional testing the program or system is treated as a black box. This box is fed with inputs and its outputs are verified with reference to a specific behavior. Thus, the software user is concerned only with the functionality of the software, but not in the implementation details. It uses the user's point of view.

In the structural testing, we make use of the implementation details. Programming methodology, control methods, source language, database design, etc. are things that matter in the structural testing. Every good system is built in layers – from outside to inside. The outer layer, which comprises only functions, is visible to the users; whereas the inner layers are less related to the functions and more to the structure. Thus, what is structural to one layer is functional to the next.

There is no controversy between structural and functional tests; both are useful and have their own shortcomings. These two tests targets have two different types of audiences. Functional tests can, ideally, detect all bugs but will take unlimited time to do so. Structural tests are finite but it cannot detect all the errors, even after they are executed completely.

1.3.3 The Designer versus the Tester

If testing was wholly based on functionality and independent of structures, then the designer and tester could work separately. However, in order to design a test plan based only on a system's structural detail would require the designer's knowledge. Thus, a designer's role is to design the transform the customer's requirement specifications into a structure which is suitable for implementation. Thus, a designer must know about the product resulting from the design and the process to obtain the same. The more information one has about the design, the better would be the designing of the tests, elimination of useless tests. Thus, the role of a tester is to verify and validate the technique that ensures that the software has been developed to meet the users' needs and specifications. Although testing can only occur after the software has been developed, the test planning and test case design activities can be conducted in parallel with the specification and design activities. A tester, thus, designs test cases, executes them and compares the actual result with the expected result to see if the software is behaving as intended. Thus, a tester's role is not structure oriented and thus allows him to work in a better way.

1.3.4 Modularity versus Efficiency

Both tests and systems can be modular. A module is a discreet, well-defined, small component of a system. Smaller components are easier to understand. Each component has multiple interfaces with which it interacts with other components. The interfaces are sources of bugs. Thus, the smaller components are more prone to interface bugs. The bigger components reduce the external interfaces but have complicated internal logic. Thus, it is important to decide the size of the components and its interface complexity to achieve an overall complexity minimization.

Testing done on smaller modules is easily repeatable. In case of an error, only the small component can be retested without the need to re-test the complete system. Similarly, if a test has an error, only one test needs to be changed and not the whole test plan.

1.3.5 Small versus Large

Building large programs means constructing different small programs by different people and then putting them together. Programming for a small code can be done single-handed, without the need of involving many people in it. With size, come into effect the non-linear scale effects. As the size of the program changes, it affects the program qualitatively and it's testing methods. Carrying out 100% testing of a small program is possible; however, a complete round of testing of an aggregate program comes down to only 75-85% and may be as low as 50% for huge systems (containing 10 millions lines of code).

1.3.6 The Builder versus the Buyer

If software is written and used by the same organization, it calls for a lot of accountability. Thus, the organizations today are shifting towards independent software development. Independent development means the developer of the software would be different from the software houses or organizations that pay for that software. This method of software development is beneficial as it leads to better software, better security and better testing. Also, it makes the accountability clear. Accountability ensures the software quality and proper software testing.

The Builder and buyer of the software can become one just as the developer and the tester. Similarly, there are other roles which can be separated or combined in a software market:

- The ***builder****,* who designs the software and is accountable for the same to.
- The ***buyer,*** who pays for the system in the hoping for profits from.
- The ***user,*** who ultimately benefits from the system ad whose interests are guarded by.
- The ***tester,*** who is dedicated to builder's destruction, and
- The ***operator,*** who lives with mistakes of the builder, shadowy specification of the buyer, oversights of the tester and the user's complaints.

Check Your Progress 1

State whether the following statements are true or false:

1. Testing is done in order to remove errors.
2. Testing requires at least half the time required to build working software.
3. Effective test cases are the ones that reveal maximum errors in minimum time.

1.4 A MODEL FOR TESTING

All the systems ranging from a subroutine to a millions of statements require testing. The typical system is that which allows exploration of all testing aspects without any complications. It's medium-scale programming. Given below in figure 1.1 is a model of testing:

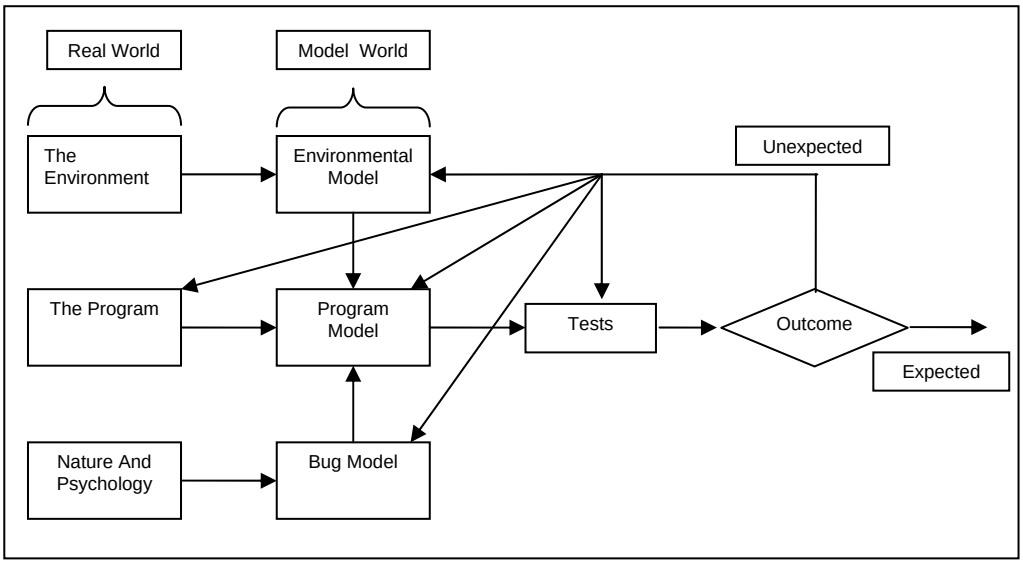


Figure 1.1: A Model for Testing

Figure 1.1 depicts a model of the testing process. It begins with a program rooted in an environment, an operating system or a calling program. Based on our understanding of the human nature and its susceptibility to error, we can create three models: model of the environment, model of the program and a model of the expected bugs. From all these models, we create a set of tests which are then executed later on. These results can either be expected or unexpected. In case of an unexpected result, we may need to change the test or the model or the way program behaves, or concept of possible bugs, or the program itself. Very rarely would it be required to change the environment.

1.4.1 The Environment

A program's environment comprises of the hardware and the software required for making it run. It also includes all the programs interacting with it and used to create this program under test e.g. operating system, loader, linker, compiler, etc.

A testing environment comprises of a test plan which details the tests that will be carried out and the test cases which describe how we will test each of the components. Creation of a test environment reduces the risk of errors occurring when the software is put to actual use and thereby reducing the downtime of the software.

1.4.2 The Program

A lot of programs are too complicated to understand. Thus, we need to simplify our concept i.e. ignore a lot of details, of the program in order to test it. We might ignore the called subroutines and concentrate on the program unless we suspect the former. Similarly, we can ignore the process details so that we can focus on the control structure of the program or vice versa. If the simple model of an environment does not explain the unexpected behavior we might have to include more facts and details.

1.4.3 Bugs

Bugs are more dangerous than we expect them to be. A lot of developers and testers have some preconceived notions about the bugs. These are just myths and need to be removed from the mind in order to test the system effectively. These are listed below:

- ***Benign bug hypothesis:*** The belief that bugs are nice, tame and logical. Only week bugs are logical and are open to exposure based on logical means. Subtle bugs have no definable patterns.

- ***Bug locality hypothesis:*** A bug occurring in a particular module affects only that module locally and not the other modules. However, this is not the case with the subtle bugs. Their consequences are far removed from the cause in time and space from the component in which they exist.
- ***Control bug dominance:*** The belief that errors in the control structure are more dominant. While bugs related to control-flow, data-flow and data structures flow can be traced easily, subtle bugs that violate the data structures boundaries and data/code separation can't be found by merely looking at the control structures.
- ***Code/Data separation:*** The belief that bugs respect the separation of data and code. The distinction between data and code is hard to make in a real system and this permits the existence of such bugs.
- ***Correction abide:*** The belief that a corrected bug remains corrected. We might have changed one of the interacting components in the event of a bug believing it as the cause, however, the bug might re-occur as it was caused by some other component which we did not change.
- ***Silver bullets:*** The mistaken belief that A (language, design, representation, environment, etc.) makes the program immune of bugs. It might reduce the severity of bugs but not the complete occurrence of bugs.
- ***Sadism effect:*** The belief that intuition and cunningness are sufficient to detect bugs. This is true for easy bugs but the tough bugs need proper methodology and techniques for detection.

1.4.4 Tests

Tests are formal procedures and are prone to bugs. Since, inputs are prepared, outputs predicted, commands executed and results observed, these steps are prone to errors. An unexpected test result could have been caused due to either test bug or real bug. Bugs can creep in documentation, inputs and the commands and over-shadow our observation of results. Thus, it might be required to change the mentality of the tester rather than the tests themselves.

1.4.5 Testing Levels

We can do different kinds of testing on a software system: unit/component testing, integration testing and system testing. Each of these methods differs in their objective and can be used in combination with one another.

- ***Unit Testing:*** A unit is the smallest testable piece of software. It is the work of a programmer and contains fewer lines of code. Unit testing is the testing done to show that the unit does not satisfy its functional specification and/or its implemented structure does not match the intended design structure. The faults resulting from such testing are called unit bugs.
- ***Component Testing:*** A component is an integration of one or more units. A unit is also a component and a component with subroutines it calls is a component, etc. By this recursive definition, a component can be anything ranging from a unit to a complete system. Component testing is the testing done to show that the component does not satisfy its functional specification and/or its implemented structure does not match the intended design structure. The faults resulting from such testing are called component bugs.
- ***Integration Testing:*** Integration is the process of aggregating the components to create larger components. Integration testing is done to show that even though the individual components were satisfactory individually, as per the components' tests, the combination of components is incorrect or inconsistent. Integration

testing is different from testing the integrated components, which is just a higher level component testing. Integration testing is aimed at exposing the problems that arise from the combination of components.

- ***System Testing:*** A system is a big component. System testing is done to reveal the bugs that cannot be attributed to the components, their inconsistencies or their interactions. System testing concerns issues and behavior that can be revealed only by testing the entire integrated system or a major chunk of it. It includes performance testing, security testing, accountability testing, configuration sensitivity testing, start-up and recovery testing.

1.4.6 The Role of Models

Testing is a process in which we create mental models of the environment, program, human nature and the tests themselves. Each model is used either till the system is accepted or until the model is no longer sufficient for the purpose. Unexpected results force the revision of the model which can be more detailed, complicated, abstract or simpler. Thus, the art of testing comprises of creating, selecting, exploring and revising models.

Check Your Progress 2

1. Fill in the blanks:
   (a) ..... testing is aimed at exposing the problems that arise from the combination of components.
   (b) ..... is the smallest testable piece of software.
2. State whether the following statements are true or false:
   (a) A bug once corrected will occur never again.
   (b) Unit testing is the smallest testable piece of software.

1.5 PLAYING POOLS AND CONSULTING ORACLES

1.5.1 Playing Pool

Testing is like a playing pool. Like the way we have a real and kiddie pool, we have real testing and kiddie testing. In kiddie testing, the tester says that the observed outcome of the test was the expected outcome. In real testing, the outcome is predicted and documented before the actual test run. If a tester cannot predict the outcome of a test, it means that the tester cannot understand the functional objectives. This misunderstanding can lead to bugs in the program or tests or both.

1.5.2 Oracles

An oracle is any program, process or body of data that specifies the expected outcome of a set of tests as applied to a tested object. There are different types of oracles like the testing concerns. The most common one, however, is the input/outcome oracle- an oracle that specifies the expected outcome for a specified input.

Sources of Oracles

If the test designer can predict the expected behavior of a test it is always good for the test design but at the same time very costly. However, this hard task has been eased with the use of oracles. Listed below are some of the sources of oracles:

1. ***Kiddie testing:*** Run the test and observe the outcome. It is always better to validate the outcome with an already available outcome rather than predicting an outcome and then validating it.

2. ***Regression test suites:*** Majority of the projects today run on rework and maintenance of the existing software. In this case, it would be required to rerun some tests that were carried on the older version on the newer version and all these should have the same outcome. Thus, it is required to predict the outcome of only the changed parts of the system.
3. ***Purchased Suites and Oracles:*** Test suites and oracles of highly standardized software are commercially available, e.g. compilers of standard languages, communication protocols, and mathematical routines. More oracles would be available in market as more and more software would get standardized.
4. ***Existing Program:*** A working entrusted program is always the best oracle. The typical use of such oracles would be like re-hosting them to a new language, OS, environment, configurations, or to a combination of these, with an intention that the behavior should not be altered after re-hosting.

1.6 IS COMPLETE TESTING POSSIBLE?

Testing carried out to show that a program is free of errors is practically and theoretically impossible. The three different approaches of testing can be structural testing, functional testing and formal proofs of correctness. Each of these approaches concludes that the complete testing is impossible.

- ***Structural Testing:*** Tests should be designed to ensure that every path of the routine is exercised at least once. Ideally, it is impossible as some of the loops might never cease. Even a small loop might have millions of paths resulting from each loop as each loop path multiplies the path count by the number of times through the loop. Thus, it can be concluded that pure structural testing cannot ensure that the system is doing the right thing.
- ***Functional Testing:*** Every program works on a finite number of inputs. A complete functional testing would mean subjecting a program to all possible inputs. For each input, the system would generate either a correct output, an incorrect output or reject the input and highlights the same. Thus, the problem is reduced only to verify that the correct outcome is produced. So, complete functional testing is practically impossible.
- ***Correctness Proofs:*** Formal proofs for correctness rely on a combination of functional and structural concepts. Requirements are stated in a formal language and each program statement is examined and used in a step of an inductive proof that the routine will produce the correct outcome for all possible input combinations. The problem here is that such proofs are very expensive and can be applied only to numerical routines or to the formal proofs for crucial software such as system's security kernel or compiler portions.

Manna and Waldinger (MANN78) have clearly summarized the theoretical barriers to complete testing:

- "We can never be sure that the specifications are correct."
- "No verification system can verify every correct program."
- "We can never be certain that a verification system is correct."

1.7 LET US SUM UP

Testing is a very critical part of software development. It consumes at least half the effort that goes into building the software. Testing is done to ensure the correctness of the program. The more errors revealed during testing, the reliable is the program. Testing ensures presence of errors but cannot guarantee the absence of errors.

A lot of techniques of testing exist like Structural, functional and correctness of proofs, however, none of these is self-sufficient to completely test the program nor can they ensure that in combination with one another.

Testing a system may require creating a model which will resemble the real environment in which the system is expected to run. Testing is like a playing pool which can be carried out with the help of oracles. Testing of a program can be carried out at different levels: unit testing, component testing, integration testing and system test.

1.8 LESSON END ACTIVITIES

1. Discuss the various testing levels in detail.
2. Is complete testing possible? Justify with proper reasoning.
3. Differentiate between testing and debugging.

1.9 KEYWORDS

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

Debugging: Debugging is the process of finding errors or misconception that led to the failure of the program and to design and implement the changes that can correct these errors.

Module: A module is a discreet, well-defined, small component of a system.

Unit: A unit is the smallest testable piece of software.

Unit testing: Unit testing is the testing done to show that the unit does not satisfy its functional specification and/or its implemented structure does not match the intended design structure.

Unit bugs: The faults resulting from unit testing are called unit bugs.

Component: A component is an integration of one or more units.

Integration: Integration is the process of aggregating the components to create larger components.

Oracle: An oracle is any program, process or body of data that specifies the expected outcome of a set of tests as applied to a tested object.

1.10 QUESTIONS FOR DISCUSSION

1. How does a designer differ from a tester? Explain.
2. Explain how modularity and efficiency related to one another.
3. Describe a testing model in detail.
4. Why is testing important? What are its goals?

Check Your Progress: Model Answers

CYP 1

1. False
2. True
3. True

CYP 2

1. (a) Integration
   (b) Unit
2. (a) False
   (b) True

1.11 SUGGESTED READINGS

Boris Beizer, *Software Testing Techniques*, Second Edition, Dreamtech Press, 2003.

Myers and Glenford, J., *The Art of Software Testing*, John-Wiley & Sons, 1979.

Roger, S. Pressman, *Software Engineering – A Practitioner's Approach*, 5th Edition, McGraw Hill, 2001.

Marnie, L. Hutcheson, *Software Testing Fundamentals*, Wiley-India, 2007.

LESSON

2

THE TAXONOMY OF BUGS

CONTENTS

2.0 Aims and Objectives

2.1 Introduction

2.2 The Consequences of Bugs

- 2.2.1 The Importance of Bugs
- 2.2.2 How Bugs Affect Us – Consequences
- 2.2.3 Flexible Severity rather than Absolute

2.3 Taxonomy for Bugs

- 2.3.1 Requirements, Features and Functionality Bugs
- 2.3.2 Structural Bugs
- 2.3.3 Data Bugs
- 2.3.4 Coding Bugs
- 2.3.5 Interface, Integration and System Bugs
- 2.3.6 Test and Test Design Bugs
- 2.3.7 Testing and Design Styles
- 2.3.8 Memory related Bugs
- 2.3.9 Concurrent Bugs

2.4 Let us Sum up

2.5 Lesson End Activities

2.6 Keywords

2.7 Questions for Discussion

2.8 Suggested Readings

2.0 AIMS AND OBJECTIVES

After studying this lesson, you would be able to understand:

- The consequences of bugs and their importance
- Requirements, features and functionality bugs
- Structural bugs and their types
- Data, coding and system bugs
- Test and test design bugs

2.1 INTRODUCTION

A bug can be important in the terms of correction cost, frequency, installation cost, and consequences. The bugs can affect us in many ways depending upon their severity. The severity must be measured in humanistic terms rather than machine terms. The importance of a bug can be calculated with the help of its frequency of occurrence, correction cost, installation and consequential cost.

A bug might be creep in any part of the software development life cycle. It could be related to requirements specification, requirements validation, requirement reviews, structural bugs, etc. We will discuss all these in detail in this lesson.

2.2 THE CONSEQUENCES OF BUGS

2.2.1 The Importance of Bugs

The importance of bugs depends upon their frequency, correction cost, installation cost and consequences.

- ***Frequency:*** It is important to know so as to how regularly a bug occurs and what are the bugs which occur most regularly.
- ***Correction cost:*** It is important to have an idea about the cost to correct the bug after it is found. This cost is the sum of two factors: (i) the cost of discovery and (ii) the cost of correction. These costs go up drastically if the bug is found in the later parts of the software life cycle. Correction of larger programs incurs higher cost.
- ***Installation cost:*** It depends upon the number of installation: small for a single user program, but how about a PC operating system bug? Installation cost can dominate over other costs – fixing one simple bug and distributing the fix could exceed the entire system's development cost.
- ***Consequences:*** The consequences of the occurrence of bugs can be measured by the means of mean size of the awards given to the victim of the bugs.

The metric for the measurement of bug importance is:

Importance ($) = Frequency* (Correction cost + Installation cost + Consequential cost)

Frequency does not tend to depend upon application or environment but the other three do. As designers, testers, and QA workers, one must be interested in bug importance and not only the frequency. Thus, it is required to come up with your own importance model.

2.2.2 How Bugs Affect Us – Consequences

The consequences of bugs range from mild to catastrophic. They should be measured in human terms and not in machine terms because programs are written to be used by the human beings. Let us discuss these consequences in detail:

1. ***Mild:*** The symptoms of the bug offend us aesthetically e.g. a misspelled output, a misaligned printout.
2. ***Moderate:*** Outputs are misleading or redundant. The bug impacts the system performance.
3. ***Annoying:*** The system's behavior is dehumanizing because of the bug. E.g. names are truncated or arbitrarily modified; bills for Rs. 0.00 have been sent, etc.

4. ***Disturbing:*** It refuses to handle legitimate transactions e.g. an ATM not dispensing money on my debit card and giving card invalid.
5. ***Serious:*** The program losing the track of transactions, the transactions occurred but the accountability is lost.
6. ***Very serious:*** Rather than losing your pay check, it is credited to other person's account.
7. ***Extreme:*** The problems are not limited to a few users or transaction types. They are frequent and arbitrary instead of erratic or for odd cases.
8. ***Intolerable:*** Long-term, unrecoverable corruption of data base, etc.
9. ***Catastrophic:*** The system fails and the decision to shut down is taken out of our hands.
10. ***Infectious:*** System corrupting other system, one that melts a nuclear reactor, etc; whose influence of malfunctioning is far more than expected; a system that kills.

2.2.3 Flexible Severity rather than Absolute

Quality of a program can be measured from the combination of number of bugs and their severity. Bugs and their symptoms play a significant role. As testing progresses we can see the quality rise from next to zero to some value at which it is deemed safe to ship the product. Let us have a look at how these parts are weighed depending upon the environment, application, culture, etc.

- ***Correction Cost:*** The cost of correcting a bug has nothing to do with the symptom severity. Catastrophic or life-threatening bugs could be easy to fix whereas minor annoying ones might require major corrections to be done.
- ***Context and Application Dependency:*** The severity of a bug may vary depending upon the context of occurrence, e.g. a round calculation may not matter much in a space video game but will matter to an astronaut.
- ***Creating Culture Dependency:*** Importance depends upon the software creators and their cultural aspirations, e.g. a test tool vendor is more sensitive towards in their production than a games software vendor.
- ***Software Development Phase:*** Severity depends on the development phase. Any bugs gets severe as it gets closer to field use and more severe the longer it is around as the correction cost of this bug increases drastically.

2.3 TAXONOMY FOR BUGS

There is no universally correct way to categorize bugs. Thus, there is no fixed taxonomy which can be applied to categorize bugs. A singe bug can be put into different categories depending upon its history and the programmer's mind state. The major categories used are: requirements, features and functionality, structure, data, implementation and coding, integration, system and software architecture and testing. Bug taxonomy is potentially infinite just like software testing. Adopting a single taxonomy and using it as a statistical framework to carry out the testing is more important than adopting a right taxonomy.

2.3.1 Requirements, Features and Functionality Bugs

Requirements and Specifications

Requirements and the specifications obtained from them can be ambiguous or incomplete. They can be difficult to understand or misunderstood. The specification

may assume other specifications which are known to the one who specifies but not to the designer. And there is also a possibility that specifications might change during design. New features would be added and old deleted or modified.

Thus, requirements are a major source of bugs. The range can vary from a few percent to more than 50% depending upon the application and environment. These bugs enter the system at the earliest and leave it the last.

Feature Bugs

Specifications problems lead to respective feature problems. A feature can be wrong, missing or superfluous. A missing feature is the easiest to detect. A wrong feature could have major design impacts. Extra features were desirable. Free features are not actually "free" in the real sense. Any increase in generality that does not lead to reliability, modularity, maintainability and robustness should be suspected. Unnecessary enhancements can lead to security breaches and at the same time one cannot forbid the additional features as they could be a result of good design.

Remedies for Specification and Feature Bugs

Human-to-human communication problems contain deep rooted feature bugs. They can be reduced by using high-level, formal specification languages or systems. These languages and systems provide short term support but can prove to be really useful in the long run.

Short-term support: Specification languages facilitate formalization of requirements and inconsistencies and ambiguity analysis. With formal specifications, partially to fully automatic test generation is possible. Users and developers of such products have found them to be cost-effective.

Long-term support: The long term impact of the formal specification languages and systems will be that they would influence the design of ordinary programming languages so that more of current specifications can be formalized. Although this approach will reduce the bugs, it will not remove them completely. These bugs would be tougher and would need even higher specification systems to be exposed.

Testing Techniques

The testing of functional bugs can be done using the functional test techniques i.e. those techniques which are based on a behavioral description of software like transaction flow testing, syntax testing, domain testing, logic testing and state testing. They are also useful to test the requirements and specifications to an extent that the requirements can be expressed in terms of the model on which the technique is based.

Check Your Progress 1

State whether the following statements are true or false:

1. Correction cost includes the cost of correcting a bug and not its detection.
2. Frequency of bugs does not depend upon application or environment.
3. The requirements and specification bugs enter the system at an early stage and leave early too.

2.3.2 Structural Bugs

Control and Sequence Bugs

Control and sequence bugs result from left-out paths, unreachable code, improper nesting of loops, and missing steps in a process, rampaging GOTO's, ill-conceived switches, etc.

Majority of software testing and design literature focuses on bugs resulting from control flows but they do not occur commonly in new software. Because this area is open to theoretical treatment, it is popular in the literature. These bugs can be tested easily and caught during unit testing.

Novice programmers cause more structural bugs than the experienced programmers. Also, old codes can be victims of more structural bugs. Thus, it is always reasonable to write new code from the scratch because the old program has become too complicated and arbitrary to be reworked upon.

Control and sequence bugs can be caught with the help of structural testing techniques like path testing combined with bottom-line functional testing based on specification.

Example:

```
i = 100;
a = 20;
If (i < a) then
   Print "i is less than a";
Else
   Goto label1:
   :
Label 2: j = i + 2;
```

The above program is an example of control bug as the control is directed to 'label 1' in the else condition, however, there is no such label in the program. Thus the program control does not know where to go in the program as a result of the else condition satisfaction.

Logic Bugs

Bugs in the logic e.g. behavior of logical statements in combination with the logic operators, include non-existent cases, improper layout of cases, improper simplification and combination of cases, overlap of exclusive cases, etc.

If these bugs are part of logical (Boolean) processing not related to control flow, then they are categorized as processing bugs. If they are a part of a logical expression which is used to direct the control flow, then they are categorized as control-flow bugs.

Logic bugs are not different from arithmetic bugs. They are likelier than arithmetic bugs because programmers have less formal training in logic at an early age than they do in arithmetic. The best defense against these bugs is a systematic analysis of cases.

Example:

```
int money, money_in_store;
int main()
{
   do
   {
    printf("Enter amount of money (0 to exit): ");
    scanf("%d", &money);
    if (money_in_store == 0)//Should be'if(money== 0)'
```

```
            {
               printf("%d money on exit\n", money_in_store);
               exit(0);
            }
            money_in_store += money;
         }
         while(1);

         return 0;
      }
```

This is an example of logical bug. After 'scanf', we are checking for 'money_in_store' instead of 'money'.

Processing Bugs

Processing bugs include arithmetic, algebraic, mathematical function evaluation, algorithm selection, incorrect conversion from one data type to another and general processing. Other problems include overflow ignoring, ignoring the difference between positive and negative zero, improper use of greater-than, greater-than-or-equal, less-than, less-than-or-equal, etc.

These bugs are frequent and can be caught during the round of good unit testing. They have localized effects. The remedies for these types of bugs include selection of covering test cases and domain testing methods.

Example:

```
int y, e, z, m, p, f, g=0, k=2, D=0, n=0, d1, d[9], x[9];
printf("Please enter decimal number followed by base number.
\n");
scanf("%d %d", &D, &n);
/* Check if entered decimal no. is already in correct base */
y = D/n;
if (y==0)
{
printf("Decimal no. %d has value of %d in base %d \n", D,D,n);

}
/* Digit-by-digit calculation of decimal no. in new base */
else
{ while(y>0){
y = D/(pow(n,k));
k+=1;
}
printf("No. of digits in final answer is %d \n", k-1);
d1 = D / (pow(n,(k-1)));
e = k-1;
f = k-1;
for (m=2, p=1; m=e, p=e-1; m++, p++)
{
z = pow(n,f);
x[p] = D%z;
```

```
f-=1;
d[m] = (x[p])/(pow(n,f));
}
printf("Decimal number of %d has value of", D);
```

The program causes a divide by zero error. ‘z’ has the values for each loop: 216->36 ->6->1->0. As ‘f’ becomes ‘-1’, the ‘pow(n,-1)’ return a Zero;

Initialization Bugs

Initialization bugs are common and both experienced programmers and testers should test them. Improper and superfluous initialization occurs. Although superfluous initialization appears to be less harmful but it can affect the performance. Typical bugs of this category are: forgetting to initialize working space, registers or data areas; a bug in the first value of a loop, wrong initialization to an improper format or data type, etc.

Example:

```
int a;
char b = ‘x’;
a = b;
```

The program above gives an initialization error as we are trying to assign the value of a character type variable to an integer variable.

The remedies to these are the kind of tools programmer has and the source language. Such bugs can be avoided by explicitly initializing all the variables being used in the program.

Data-flow Bugs and Anomalies

Most of the initialization bugs are special cases of the data flow anomalies. A data flow anomaly occurs when there is a path along which we expect to do something unreasonable with data e.g. using an uninitialized variable, attempting to use a variable before it exists, modifying data and not storing or using the result, etc. Although part of such anomalies can be detected by the compile time information much can be achieved only by executing and subjecting to tests.

2.3.3 Data Bugs

Data bugs are those which arise from the specification of data objects, their formats, the number of such objects and their initial value. They are as common as the code bugs but are often ignored. They are underestimated because of poor bug accounting. Because data bugs are not counted in some situations, the parts of the program which include data declaration statements are also ignored. The separation of code and data is artificial as they can be used interchangeably. In the extreme case, a single 20 line program can perform the task of a any computer and have all programs recorded as data and manipulated as such.

Software is evolving towards programs in which more and more of the control and processing functions are stored in tables. This is called the third law.

Third Law – Code migrates to data

Because of this law, the awareness about the bugs in the code has increased. There is an awakening that the bugs in the code are only half the battle and that the data problems are equally important. Each of these has bug has a frequency of 25%.

Data declaration statements are equally prone to cause errors even though they are not executed and are specified by humans.

The increase in the proportion of the source statements devoted to data definition is a direct consequence of the following: (1) the dramatic reduction in the cost of main memory and disc storage, and (2) the high cost of crating and testing software. Generalized software is not efficient. The increase in the cost of software as a percentage of system cost has resulted in shifting the emphasis from single-purpose, unique software to an increased reliance on a pre-packaged generalized program. These programs must satisfy a wide range of options, operating systems, computers, etc. This generalization is achieved by making the program more parameterized and then setting the values of these parameters for specific installations.

Another source of increase in the database complexity is the use of control tables in place of code. Instead of being coded in the form of computer instructions or language statements, the steps required to process a transaction are stored as a sequence of constants in a transaction-processing table. The state of the currently executing transaction is stored in a transaction-control block. The combination of these two is used in the generalization of the transaction control process. The transaction-control table is a program which is processed interpretively by the transaction-control processor. In other words, a hidden programming language has been created.

The first step in the avoidance of data bugs – whether the data is a pure data, parameters or hidden – is the realization that all source statements, including data declarations, must be counted, and that all source statements, whether or not they result in object code, are bug-prone.

Dynamic versus Static

Dynamic data is momentary irrespective of their purpose. Their lifetime is normally limited to the processing time of a transaction. They are stored in the storage objects. In case a shared object is not initialized properly, it can lead to data-dependent bugs caused by the residues from a previous use of that object by another transaction. Here, the culprit transaction is long gone before the bugs are discovered. Thus, the effects of a corrupt dynamic data can be far reaching and can be difficult to catch. The design remedy is the complete documentation of all shared memory structures, defensive code that goes through data validation checks, and centralized-resource managers.

The basic problem is leftover garbage in a shared resource. This can be taken care of in any one of the following three ways: (1) cleanup after use by the user, (2) common cleanup by the resource manager, and (3) no cleanup.

Static data is fixed in form and content. They appear in the source code or database directly or indirectly irrespective of their purpose. It need not be explicit in the code. Some languages have compile-time processing which is useful for general purpose routines that are particularized by interrelated parameters. It is an effective measure against parameter-value conflicts. Rather than relying on the programmer to calculate correct values of interrelated parameters, a program executed at the compile time takes care of it. If compile-time processing is not a language feature, a specialized processor can be built which calculates the parameter values.

Another example is the preprocessing (or post-processing) code, any code executed at compile time or assembly time or before, at load time, at installation time, or some other time can lead to faulty static data and therefore bugs – even though such code does not represent object code at run time.

The design remedy for the preprocessing situation is present in the source language, if it permits compile-time processing. This will eliminate the need of a specialized processor.

Information, Parameter and Control

Static or dynamic data can serve in one role or a combination of roles: as a parameter, for control or for information. Constitution of control and information depends upon the perspective and can shift from one level to the other.

Information is dynamic and tends to be local to a single transaction or task. Errors in information may not be serious bugs. The bugs could be a result of lacking data validations, failure to protect the code logic from data out of range or data in an incorrect format. Data validation related bugs can be avoided only by ensuring that the validations are included in the code. Not implementing data validation in a routine thinking that it would be done in another routine can often lead to forgetting to implement the validations at all. The program is developed and changed without remembering that the modified routine did the data validation for several other routines. Blocking the vulnerable data in a routine can save it from bad data and at the same time it's even better to redesign the routine so that it does not remain vulnerable any more.

Inadequate data validations lead to finger-pointing wherein the writer of calling routine often blames the writer of the called routine and vice versa. And they both together blame the operator. This attitude is understandable but not correct. Thus, even though a fellow programmer did thorough and correct validations, static data, parameters and code can get corrupted.

Contents, Structure and Attributes

Data specification comprises of three parts:

- ***Contents:*** These are actual bit patterns, character string or number put into a data structure. Content is a pure bit pattern and holds no meaning unless it is interpreted by a hardware or software processor. All the data bugs result in the corruption or misinterpretation of content.
- ***Structure:*** It is the size, shape and numbers which describe the data object, i.e. the memory location used to store the contents. Structures can have sub-structures and can be made into super-structures.
- ***Attributes:*** They are the semantics associated with the contents of a data object i.e. the specification of the meaning e.g. an integer, subroutine, alphanumeric string, etc.

2.3.4 Coding Bugs

Coding bugs of or more kind can lead to bugs of the other kinds. Syntax errors are normally taken care of by themselves if the language translator has adequate syntax checking. A good translator can catch bugs related to undeclared data, undeclared routines, dangling code and other initialization problems. Programming errors caught by the translator does not affect the test design and execution drastically because testing cannot begin unless these errors are rectified. But a program having multiple source-syntax errors can have many logic and coding bugs.

Coding bugs are wild-card of programming unlike the logic or process bugs which have their own perverse rationality. Wild-cards are arbitrary. The most common kind of coding bugs are documentation bugs which are often ignored considerably. Documentation bugs could be as simple as spelling mistakes, misleading or erroneous comments, etc. Although they appear to be small but they cannot be ignored because its consequences could be as big as any coding errors. Such bugs can also lead to incorrect maintenance actions and thus cause insertion of other bugs.

Example:

In the user manual of MS Word, a line in the “How to insert an image from your hard disk in a word document” reads

“To insert an image inside a word document, click on Insert -> Pictures -> from internet”

However, actually it must read

“To insert an image inside a word document, click on Insert -> Pictures -> from file on disk”

This can cause a confusion to a new user who is using this feature of the MS word for the first time.

Thus, one needs to be very careful while creating documents and manuals pertaining to software and products for its users.

2.3.5 Interface, Integration and System Bugs

External Interfaces

External interfaces are the means to communicate with the outside world. These could be devices, actuators, sensors, input terminals, printers and communication lines. The primary design criterion for interfaces should be its robustness. These interfaces must employ a protocol. Protocol could be complicated and difficult to follow. Protocol itself can be wrong if it’s new or could be wrongly implemented. Other external interface errors are misunderstanding external input and output formats, insufficient tolerance to bad input data, etc.

Internal Interfaces

In principle, internal interfaces are not different from external interfaces but they do differ in practice as the internal environment is more controlled. The external environment is fixed and the system must adapt to it but the internal environment can be negotiated as it consists of interfaces to other components as well. Internal interfaces can have the problems of the external interfaces as well of their own like those related to the implementation details. The real remedy is the design and in standards. They must not be allowed to grow just like that. They should be formal and as few as possible. The number of different internal interfaces can be traded-off with the complexity of a single internal interface.

Hardware Architecture

Software bugs related to hardware crop up from the misunderstanding of how does the hardware works. Examples of such bugs are as follows: I/O device operation or instruction error, I/O device address error, wrong format expected, data format wrong for device, device protocol error, waiting too long for a response, incorrect interrupt handling, ignoring hardware fault or error condition, ignoring operator malevolence, assuming that the device has been initialized.

The remedy for hardware architecture and interface problems is two fold: (1) good programming and testing and (2) centralization of hardware interface software in programs written by hardware interface specialists. Modern hardware is difficult to test as it lesser buttons, switches and lights but cheaper. This paradox can be resolved by hardware that has special test modes and test instructions that do what the buttons and switches do. However, such features are yet to be provided by the hardware manufacturers. Or it could also be advisable to use a hardware simulator instead of the actual hardware.

Operating System

Program bugs related to the operating system are a combination of hardware architecture and interface bugs, mostly caused by the misunderstanding of what the operating system does. And an OS could have the bugs of its own. Often an OS may lull the programmer that the hardware interface errors have been addressed by it. As the operating system gets old its bugs are found and corrected but some of these corrections may leave peculiarities.

The remedy for the OS interface bugs is the same as for hardware bugs; use OS interface specialists and use explicit interface modules or macros for all operating system calls. This approach may not eliminate all bugs but at least will localize them and make testing easier.

Software Architecture

Software architecture bugs are those which are often termed as “interactive”. Routines can pass the unit and integration testing without revealing such bugs. Many of them depend on load and their symptoms emerge only when the system is stressed. They tend to be the most difficult kind of bugs to find unearth. Examples of the causes of such bugs are: failure to open or close an interlock, assumption that a called routine is resident or not resident, assumption that registers or memory were initialized or not initialized, assumption that register or memory location content did not change, local settings of global parameters and global setting of local parameters.

The first line of defense against these bugs is the design for the software architecture. All testing techniques are applicable to the discovery of software architecture bugs, but experience has shown that careful integration of modules and subjecting the final system to a brutal stress test are especially effective.

Check Your Progress 2

1. Fill in the blanks:
   (a) Control and sequence bugs can be caught with the help of ..... testing.
   (b) Logic bugs are similar to ..... bugs.
2. State whether the following statements are true or false:
   (a) Processing bugs do not have localized effects.
   (b) It is not easy to separate data and code.

Control and Sequence Bugs

System level control and sequence bugs include: ignored timing, assuming that events occur in a specified sequence, starting a process before its prerequisites are met, waiting for an impossible prerequisite to be met, specifying a wrong priority, program state or processing level, etc.

The remedy for these bugs is in the design. Highly structured sequence control is helpful. Specialized, internal, sequence – control mechanisms, such as internal job control language, are useful.

Integration Bugs

Integration bugs are those that come into being with the integration of and with the interfaces between, presumably working and tested components. Most of these bugs result from inconsistencies or incompatibilities between components. These bugs are hosted in the methods used to transfer data directly or indirectly between components and all methods by which components share data. The communication methods

include data structures, call sequences, registers, semaphores, communication links, protocols, etc. Although these bugs do not constitute a big category of bugs but are an expensive category because they are caught late and force changes in other components. These bugs can be revealed using domain testing, syntax testing and data-flow testing.

System Bugs

System bug comprises of all those bugs that cannot be ascribed to components or to their simple interactions, but result from the totality of interactions between many components like programs, data, hardware and the OS. These can be revealed with the help of transaction-flow testing techniques but two factors however, should be kept in mind: (1) all test techniques can be useful at all levels, from unit to system and (2) there can be no meaningful system testing until there has been thorough component and integration testing. System bugs are not frequent but very expensive as they are caught after the system has been fielded and because the fix is rarely simple.

2.3.6 Test and Test Design Bugs

Testing

Testers are not immune to bugs. System tests require complex scenarios and databases, code or equivalent to execute and thus, can have bugs. The virtue of independent functional testing is that it provides an unbiased point of view but this provides an opportunity for different and possibly incorrect interpretations of the specifications. Test bugs are not software bugs but are difficult to be separated from them.

Test Criteria

Test criteria are used to judge whether the software's behavior is incorrect or impossible. The more complicated the criterion the likelier it is to have bugs.

Remedies

The remedies for test bugs are; test debugging, test quality assurance, test execution automation and test design automation.

- ***Test Debugging:*** The first remedy for test bugs and debugging is testing and debugging the tests. Test debugging is usually easier because tests when properly designed are simpler than programs and do not have to make concessions to efficiency. Also, tests tend to have a localized impact relative to other tests and therefore the complicated interactions that usually plague software designers are less frequent.
- ***Test Quality Assurance:*** Programmers have the right to ask how quality in independent testing and test design is monitored. Should we implement test testers and test-tester tests? This sequence does not converge.
- ***Test Execution Automation:*** Test execution bugs are virtually eliminated using various test execution automation tools. Manual testing is self-contradictory. Bugs cropping up due to manual testing can be reduced by using test execution automation.
- ***Test Design Automation:*** For a given productivity rate, automation reduces bug count – be it for software or be it for tests.

2.3.7 Testing and Design Styles

Bad designs lead to bugs and are difficult to test; therefore, the bugs remain. Good designs inhibit bugs before they occur and are easy to test. The two factors are multiplicative, which explains the productivity difference. The best test techniques are useless when applied to abominable code; it is sometimes easier to redesign a bad routine than to attempt to create tests for it. The labor required producing new code and design for the new code is much lesser than the labor required to design thorough tests for an undisciplined, unstructured monstrosity. Good testing works best on good code and good designs.

2.3.8 Memory related Bugs

Memory related bugs are caused due to improper handling of the memory objects. These bugs can be used to cause security breaches in software. These bugs are the biggest contributors to all vulnerabilities of software. They can be further classified as under:

- ***Buffer overflow bugs***: They are caused when we try to access the allocated memory beyond the buffer boundary.
- ***Stack smashing***: These bugs are caused when we illegally overwrite the function return address.
- ***Memory leak***: These bugs are caused when we are not able to access a dynamically allocated memory to free it.
- ***Uninitialized read***: These bugs occur when we try to read a memory data before it is initialized.
- ***Double free***: These bugs occur when we try to free an already freed memory location.

Example:

```
int main(void)
{
     /* this is an infinite loop calling the malloc
function which
      * allocates the memory but without saving the
address of the
      * allocated place */
    while (malloc(50)); /* malloc will return NULL
   sooner or later, due to lack of memory */
     return 0; /* Did not free the allocated memory at
all before returning */
}
```

The C function above deliberately leaks memory by losing the pointer to the allocated memory. Since the program loops forever calling the defective function, malloc(), but without saving the address, it will eventually fail (returning NULL) when no more memory is available to the program. Because the address of each allocation is not stored, it is impossible to free any of the previously allocated blocks.

2.3.9 Concurrent Bugs

Concurrent bugs are those that can occur only in multi-threading or multi-process environment. They are caused because of the bad synchronization of the operations of multiple threads. These bugs are un-deterministic in nature which makes them

difficult to reproduce. Such temporary sensitivity makes the bug detection more difficult. They can be classified as below:

- ***Datarace bugs:*** They occur because of the conflicting access from concurrent threads when they try to use the same shared memory in different orders.
- ***Atomicity related bugs:*** These are caused when a bunch of operations from a single thread are unexpectedly interrupted by a conflicting operation from some other threads.
- ***Deadlock:*** In resource sharing, one or more processes wait indefinitely for some resource to get free and cannot proceed further, thus, leading to a deadlock.

Example:

```
object lockA = new object();
object lockB = new object();

// Thread 1
void t1() {
  lock (lockA) {
    lock (lockB) {
      /* ... */
    }
  }
}

// Thread 2
void t2() {
  lock (lockB) {
    lock (lockA) {
      /* ... */
    }
  }
}
a
```

To further illustrate how a deadlock might occur, imagine the following sequence of events:

- Thread 1 acquires lock A.
- Thread 2 acquires lock B.
- Thread 1 attempts to acquire lock B, but it is already held by Thread 2 and thus Thread 1 blocks until B is released.
- Thread 2 attempts to acquire lock A, but it is held by Thread 1 and thus Thread 2 blocks until A is released.

At this point, both threads are blocked and will never wake up.

2.4 LET US SUM UP

The importance of a bug is dependent on its occurrence frequency, the correction cost, the consequential cost and the application. Thus, the testing resources must be allocated in proportion to the importance of the bugs.

Effectiveness of the testing techniques depends upon the target. The test techniques must be best suited to the kind of bugs you have. The effectiveness of the test techniques erodes with time. A comprehensive bug classification is a prerequisite to gathering useful bug statistics. Adopt taxonomy, simple or elaborate, but adopt one and classify all bugs within it.

2.5 LESSON END ACTIVITIES

1. What are control and sequence bugs and how are they minimized?
2. How are the internal interface bugs different from external ones?
3. Define contents, structure and attributes in the light of data specification.
4. What is the third law in context of data bugs? Explain in detail.

2.6 KEYWORDS

OS: Operating System

External interfaces: External interfaces are the means of a program to communicate with the outside world.

Data bugs: Data bugs are those which arise from the specification of data objects, their formats, the number of such objects and their initial value.

Data flow anomaly: A data flow anomaly occurs when there is a path along which we expect to do something unreasonable with data.

2.7 QUESTIONS FOR DISCUSSION

1. Explain the structural bugs in detail mentioning all the different bugs which form part of it.
2. Discuss the various bug consequences in detail.
3. How can we measure the importance of a bug? Give a mathematical formula to measure the same.

Check Your Progress: Model Answers

CYP 1

1. True
2. False
3. False

CYP 2

1. (a) structural
   (b) arithmetic
2. (a) False
   (b) True

2.8 SUGGESTED READINGS

Boris Beizer, *Software Testing Techniques*, Second Edition, Dreamtech Press, 2003.

Myers and Glenford, J., *The Art of Software Testing*, John-Wiley & Sons, 1979.

Roger, S. Pressman, *Software Engineering – A Practitioner's Approach*, 5th Edition, McGraw Hill, 2001.

Marnie, L. Hutcheson, *Software Testing Fundamentals*, Wiley-India, 2007.

UNIT II

LESSON

3

SOFTWARE TESTING TECHNIQUES

CONTENTS

3.0 Aims and Objectives

3.1 Introduction

3.2 Testing Fundamentals

- 3.2.1 Objectives of Testing
- 3.2.2 Benefits of Testing
- 3.2.3 Testing Principles

3.3 Test Case Design

3.4 White-box Testing

3.5 Black-box Testing

- 3.5.1 Boundary Value Analysis
- 3.5.2 Equivalence Class Testing
- 3.5.3 Decision Table-based Testing
- 3.5.4 Cause Effect Graphing Technique

3.6 Let us Sum up

3.7 Lesson End Activities

3.8 Keywords

3.9 Questions for Discussion

3.10 Suggested Readings

3.0 AIMS AND OBJECTIVES

After studying this lesson, you would be able to understand:

- Testing objectives, principles and testability
- How to design the test cases
- How to conduct white-box and black-box testing

3.1 INTRODUCTION

Software testing is a critical element of software quality assurance and resents the final review of specification, design, and code. The source-code once generated needs to be tested for bugs and defects to get rid of maximum errors before the software is sent to the customer. Test cases are designed with an aim to find errors.

During initial stages of testing, a software engineer performs all the tests. However, at later stages a specialist may be involved. Tests must be conducted to find the highest possible number of errors, must be done systematically and in a disciplined way. Testing involves checking both the internal program logic and the software requirements.

3.2 TESTING FUNDAMENTALS

Testing software can be considered as the only destructive (psychologically) step in the entire life cycle of software production. Although all the initial activities aimed at building a product, the testing is done to find errors in software.

3.2.1 Objectives of Testing

- Executing a program in order to find errors.
- A good test case is the one that has a high probability of finding an undiscovered error.
- A successful test is the one that reports an as–yet undiscovered error.

The objective while designing tests is to come up with the test cases that systematically uncover different classes of errors in minimum possible time and effort.

3.2.2 Benefits of Testing

- It reveals the errors in the software.
- It ensures that software is functioning as per specifications and it meets all the behavioral requirements as well.
- The data obtained during testing is indicative of software reliability and quality as a whole.
- It indicates presence of errors and not absence of errors.

3.2.3 Testing Principles

Before coming up with ways to design efficient test cases, one must understand the basic principles of testing:

- ***Test cases must be traceable to requirements:*** Because software testing reveals errors, so, the severe defects will be those that prevent the program from acting as per the customer's expectations and requirements.
- ***Test planning must be done before beginning testing:*** Test planning can begin soon after the requirements specification is complete and the detailed test cases can be developed after the design has been fixed.
- ***Pareto principle applies to software testing:*** Pareto principle states that 80 percent of the uncovered errors during testing will likely be traceable to 20 percent of all the program components. Thus, the main aim is to thoroughly test these 20 percent components after identifying them.
- ***Testing should begin with small and end in large:*** The initial tests planned and carried out are usually individual components and as testing progresses the aim shifts to find errors in integrated components of the system as whole rather than individual components.
- ***Exhaustive testing is impossible:*** For a normal sized program the number of permutations of execution paths is very huge. Thus, it is impossible to execute all

the combinations possible. Thus, while designing a test case it should be kept in minds that it must cover the maximum logic and the component-level design.

- ***Efficient testing can be conducted only by a third party:*** The highest probability of finding errors exists when the testing is not carried by the party which develops the system.

A program developed should be testable i.e. it should be possible to test the program. The testability of a program can be measured in terms of few properties like: operability, observability, controllability, decomposability, simplicity, stability, understandability, etc.

The test also, must also comply with the characteristics of a good test case. These characteristics are mentioned here:

- The probability of finding error should be high. In order to achieve this, tester must understand the actual functionality of the software and think of a suitable condition that can lead to failure of the software.
- A test case must be non-redundant. Because the testing time and resources are limited, it will waste a lot of time to conduct the same test again and again. Every test must have a distinct purpose.
- A test case must be of the best quality. In a group of similar test cases, the one that covers the maximum scenarios to uncover maximum errors should only be used.
- A good test case should neither be too simple nor too complex. A complex test that includes several other tests can lead to masking of errors. Thus, each test case should be executed separately.

3.3 TEST CASE DESIGN

The designing of tests for software can be as challenging as the initial design of the product itself. Thus, software engineers must design test cases that have highest probability of finding defects within minimum time and effort.

A lot of methods exist to design test cases. These methods ensure systematic development of test cases, completeness and highest likelihood of finding errors.

A product can be tested in two ways:

1. Knowing the specified function that a product has been designed to perform; tests can be done to ensure that these functions are existing and that too without errors.
2. Knowing the internal workings of a product the tests can be performed to ensure that all the internal operations are being performed as per specifications and all internal components have been adequately built.

The first approach is called black-box testing and the second, white-box testing.

Black-box testing of software refers to tests that are conducted at the software interfaces. These kinds of tests ensure that software functions are operational, input is accepted properly and output is correctly produced and that the external information e.g. database is maintained. It does not deal in depth with the internal logic of the program.

White-box testing works on the principle of closely monitoring the procedural details of the software. All the logical paths of the software are tested using test cases for specific conditions and loops. The actual results are compared with the expected results to ensure that the correct results are produced as a result of the functions being performed.

The quality of the test cases or their appropriateness should be such that it can check the quality of the system being tested. Thus, it requires clearly distinguishing of the object that must be tested and structuring the different dimensions of test case quality using the available documents and artifacts from different phases of the development process. The object being considered for quality evaluation can be either a single test case or a set of test cases i.e. test suite. The quality assessment of a single test case would be different from that of a test suite.

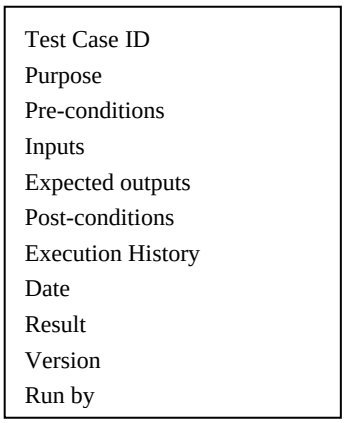


Figure 3.1: Typical Test Case Information

A typical test case must have the attributes defined in Figure 3.1 for its identification. The test case ID allows us to identify the test case uniquely, purpose defined the objective of the test case, preconditions refer to the conditions that must be satisfied beefier the test case is actually satisfied, inputs refer to the inputs which should be made to the test case at the time of execution, expected output is the output generated as a result of execution of test case which conforms to the software design. Fields like date, version and 'run by' are generic in nature for all the test cases belonging to a particular test suite.

Check Your Progress 1

State whether the following statements are true or false:

1. Testing indicates presence of errors and not absence of errors.
2. A test case must be designed to cover the minimum logic and component-level design.

3.4 WHITE-BOX TESTING

White-box testing or glass-box testing is a test case design method that uses the control structure of the procedural design to obtain test cases. Using this methodology, a software engineer can come up with test cases that:

1. Guarantee that all independent paths within a module have been exercised at least once
2. Exercise all the logical decisions based on their true and false sides
3. Executing all loops at their boundaries and within them
4. Exercise internal data structures to ensure their validity.

The reason of conducting white-box testing largely depends upon the nature of defects in software:

- Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed.
- We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis.
- Typographical errors are random.

Each of these reasons provides an argument for conducting white-box tests. Black-box testing might miss out these kinds of errors. A number of methods are associated with structural testing which are discussed in detail in section 3.5.

3.5 BLACK-BOX TESTING

Black-box testing or behavioral testing, mainly focuses on the functional requirements of the software. It enables the software engineer to develop a set of inputs that can fully involve all the functional requirements for a program. It is not an alternative approach to white-box testing. Rather, it completes the testing by uncovering different types of errors than the white-box approach.

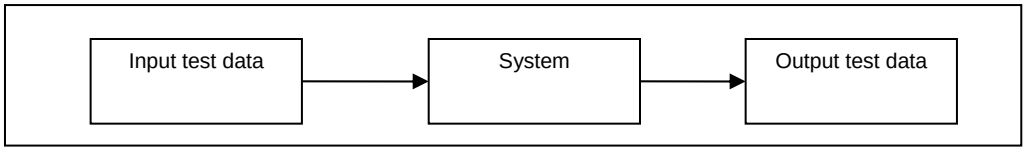


Figure 3.2: Black-box Testing

As shown in Figure 3.2, in black-box testing, the software is considered as a black box whose internal structure is not known. This box is fed with relevant inputs to generate the output data. This output data obtained is then checked against the design to ensure that it is as per software specification.

This approach attempts to find errors in the following categories:

1. Incorrect or missing functions
2. Interface related errors
3. Errors in data structures or database accesses
4. Performance related errors
5. Initialization and termination errors.

This technique is applied at a later stage unlikely to the white-box testing approach. Using this technique, we can arrive at test cases that satisfy the following criteria:

1. Test cases that reduce the total test cases by a large quantity to achieve reasonable testing,
2. Test cases that tell us something about the presence or absence of a variety of errors rather than a particular kind of error.

The techniques are as below:

3.5.1 Boundary Value Analysis

The art of testing is to come up with a small set of test cases such that the chances of detecting an error are maximized while minimizing the chances of creating redundant test cases that uncover similar errors. It has been observed that the probability of finding errors increases if the test cases are designed to check boundary values.

Consider a function F with x and y as two input variables. These inputs will have some boundaries:

a<=x<=b

p<=y<=q

Hence, inputs x and y are bounded by two intervals [a,b] and [p,q] respectively. For x, we can design test cases with values a and b, just above a and b and just below a and b which will have higher chances to detect errors. Similar is the case for y. This is represented in the Figure 3.3.

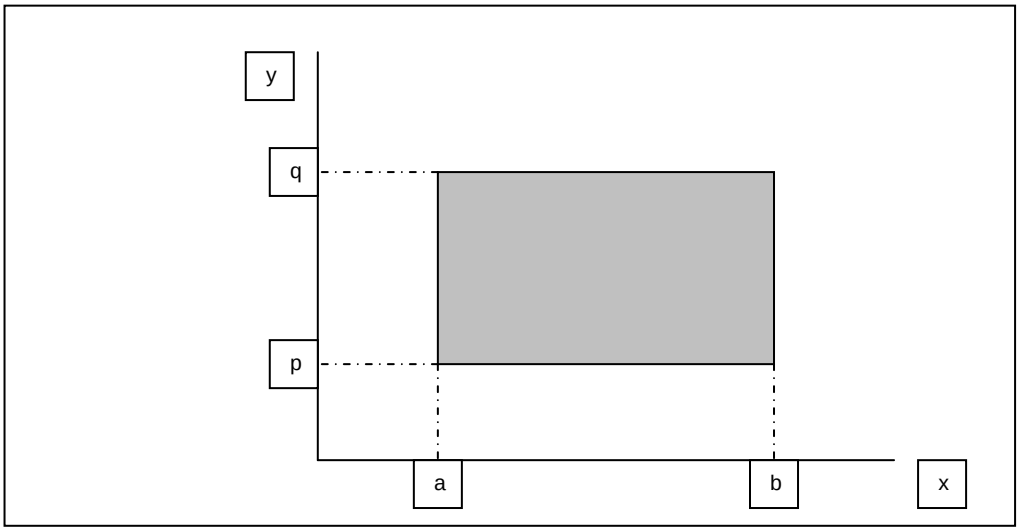


Figure 3.3: Input Domain of a Function of two Variables

Boundary value test cases are obtained by keeping one variable at its extreme and the other at its nominal value. At times we might want to check the behavior when the value of one variable exceeds its maximum. This is called robustness testing. We may also include test cases to check what happens when more than one variable have maximum values. This is called worst case analysis. Using prior testing knowledge and experience to test similar programs is called ad-hoc testing.

3.5.2 Equivalence Class Testing

In this method of testing, input domain of a program is divided into a finite number of equivalence classes such that the test of a representative value of each class is equivalent to a test of any other value, i.e. if a test in a class detects one error all other test cases belonging to the same class must detect the same error. Also, if a test case in a class did not detect an error the other test cases of the same class also should not detect the error. This method of testing is implemented using the following two steps:

1. The equivalence class is identified by taking each input condition and dividing it into valid and invalid classes.
2. Developing test cases using the classes identified. This is done by writing test cases covering all the valid equivalence classes and a single case for invalid equivalence class.

Again good test cases will be those that check for boundary value condition.

Check Your Progress 2

1. State whether the following statements are true or false:
   (a) Using White-box testing, a software engineer can come up with test cases that guarantee that all independent paths within a module have been exercised at least once.
   (b) Black-box testing is an alternative approach to white-box testing.
2. Fill in the blanks
   (a) ……..…… of software refers to tests that are conducted at the software interfaces.
   (b) White-box testing is also known as …….
   (c) Using prior testing knowledge and experience to test similar programs is called ……

3.5.3 Decision Table-based Testing

Decision tables are useful for describing situations in which a number of combinations of actions are taken under varying sets of conditions. There are four parts of a decision table namely, Condition stub, Action stub, Condition entries and Action entries. These are described in Figure 3.4.

Condition Stub	Entry						
C1	True				False		
C2	True		False		True		False
C3	True	False	True	False	True	False	-
Action Stub A1	X	X			X		
A2	X		X			X	
A3		X			X		
A4				X		X	X

Figure 3.4: Decision Table Terminology

The decision table given in Figure 3.4 has three conditions C1, C2 and C3 each of which could be either true or false. Based upon the values of these conditions, the actions A1, A2, A3 and A4 would be taken. The combination of the values of the conditions C1, C2 and C3 decide the action that would be taken as a result.

To develop test cases from decision tables, we treat conditions as input and actions as outputs. Sometimes conditions end up referring to equivalence classes of inputs, and actions refers to major functional processing portions of the item being tested.

3.5.4 Cause Effect Graphing Technique

One drawback of boundary value analysis and equivalence partitioning is that these they do not explore combinations of input circumstances which may result in interesting conditions. These situations must be tested. If we consider all possible valid combinations of equivalence classes, then it results in a large number of test cases, many of which may not uncover any undiscovered errors.

This technique helps in selecting, in a systematic approach, a high-yield set of test cases. It is also useful in pointing out incompleteness and ambiguities in the specifications. The following steps are used to derive test cases:

1. The causes and effects are identified. A cause is a distinct input condition and effects are output conditions or a system transformation. These are identified by reading the specification and identifying the words or phrases that describe causes and effects. Each cause and effect is assigned a unique number.
2. The semantic content of specification is studied and transformed into a Boolean graph linking the causes and effects. This is the cause effect graph.
3. The graph is annotated with constraints describing combinations of causes and/or effects that are impossible because of syntactic or environmental constraints.
4. By methodically tracing state conditions in the graph, the graph is converted into a limited entry decision table. Each column in the graph represents a test case.
5. The columns in the decision table are converted into test cases. The basic notation for the graph is shown in Figure 3.5.

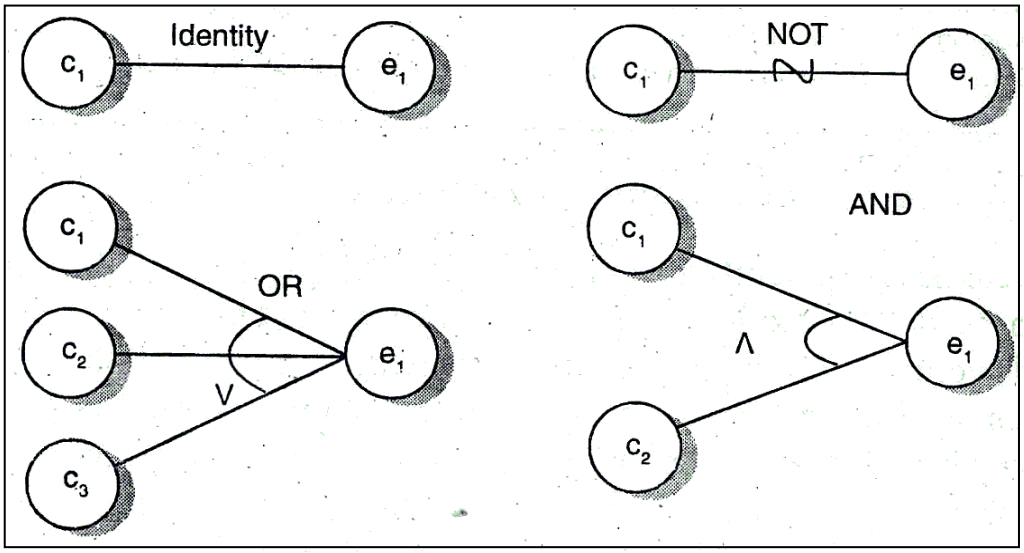


Figure 3.5: Basic Cause Effect Graph Symbols

As given in Figure 3.5, the cause-effect symbol for an identity is such that corresponding to a cause there would be one effect. In case of a not, the effect would be the inverted value of the cause, i.e. for $c_1 = 0$, $e_1 = 1$ and for $c_1 = 1$, $e_1 = 0$. For an OR cause-effect graph, the value of e_1 would be either of the input causes, i.e. c_1, c_2 and c_3 in this case. Even if one of the values of c_1, c_2 or c_3 is 1, e_1 would be 1. For the cause-effect graph of AND, the value of e_1 is the result of c_1 AND c_2, i.e., for $e_1 = 1$ both c_1 and c_2 necessarily have to be 1. If either of c_1 or c_2 is 0, e_1 would result in 0.

Think of each node as having values either 0 or 1, where 0 represents the 'absent state' and 1 represents the 'present state'. The identity function states that if c_1 is 1, e_1 is 1; else e_1 is 0. The NOT function states that if c_1 is 1, e_1 is 0 and vice versa. The OR function states that if c_1 or c_2 or c_3 is 1, e_1 is 1 else e_1 is 0. The AND function states that if both c_1 and c_2 are 1, e_1 is 1 else e_1 is 0. The AND and OR functions can have any number of inputs.

3.6 LET US SUM UP

The primary objective of test case design is to derive a set of tests that have the highest probability for fining errors in the software. To accomplish this, two different categories of test case design are possible: white-box testing and black-box testing.

White-box tests focus on the program control structure. Test cases are developed ensuring that all the statements in the program are executed at least once while testing and that all logical conditions are checked. It makes use of various techniques like path testing, condition and data flow testing.

Black-box tests are designed to validate the functional requirements without considering the internal workings of a program. It focuses on the information domain of the software, deriving test cases by dividing input and output domain of a program in such a way that is provides 100% coverage. The various techniques used are equivalence partitioning, boundary value analysis, etc.

3.7 LESSON END ACTIVITIES

1. What are the objectives of testing? Why is the psychology of a testing person important?
2. What drawbacks of boundary value analysis and equivalence partitioning are covered by cause effect graphing technique?

3.8 KEYWORDS

Robustness testing: Checking the behavior when the value of one variable exceeds its maximum is called robustness testing.

Worst case analysis: Testing by including test cases to check what happens when more than one variable have maximum values is called worst case analysis.

Ad-hoc testing: Testing using prior testing knowledge and experience to test similar programs is called ad-hoc testing.

Black-box testing or behavioral testing: Black-box testing of software refers to tests that are conducted at the software interfaces.

White-box testing or glass-box testing: White-box testing is a test case design method that uses the control structure of the procedural design to obtain test cases.

Pareto principle: Pareto principle states that 80 percent of the uncovered errors during testing will likely be traceable to 20 percent of all the program components.

3.9 QUESTIONS FOR DISCUSSION

1. Does exhaustive testing ensure that a program is 100% correct?
2. Discuss the difference between worst case and ad-hoc test case performance evaluation method of testing.
3. Differentiate between black-box testing and white-box testing.
4. List down the various testing principles.

Check Your Progress: Model Answers

CYP 1

1. True
2. False

Contd….

CYP 2

1. (a) True
   (b) False
2. (a) Black-box testing
   (b) Glass-box testing
   (c) Ad-hoc testing

3.10 SUGGESTED READINGS

Boris Beizer, *Software Testing Techniques*, Second Edition, Dreamtech Press, 2003.

Myers and Glenford, J., *The Art of Software Testing*, John-Wiley & Sons, 1979.

Roger, S. Pressman, *Software Engineering – A Practitioner's Approach*, 5th Edition, McGraw Hill, 2001.

Marnie, L. Hutcheson, *Software Testing Fundamentals*, Wiley-India, 2007.

LESSON

4

FLOWGRAPHS AND PATH TESTING

CONTENTS

4.0 Aims and Objectives

4.1 Introduction

4.2 Path Testing Basics

- 4.2.1 Motivation and Assumption
- 4.2.2 Control Flow-graphs
- 4.2.3 Notational Evolution
- 4.2.4 Path Testing
- 4.2.5 Loops
- 4.2.6 Variations

4.3 Predicates, Path Predicates and Achievable Paths

- 4.3.1 Predicates
- 4.3.2 Predicate Expressions
- 4.3.3 Predicate Coverage
- 4.3.4 Testing Blindness

4.4 Path Sensitizing

4.5 Path Instrumentation

- 4.5.1 Problem
- 4.5.2 Link Markers
- 4.5.3 Link Counters
- 4.5.4 Implementation

4.6 Instrumentation and Application of Path Testing

- 4.6.1 Integration, Coverage and Paths in Called Components
- 4.6.2 New Code
- 4.6.3 Maintenance
- 4.6.4 Re-hosting

4.7 Let us Sum up

4.8 Lesson End Activities

4.9 Keywords

4.10 Questions for Discussion

4.11 Suggested Readings

4.0 AIMS AND OBJECTIVES

After studying this lesson, you would be able to understand:

- Basics of path testing – motivation and assumptions, loops, control flow-graphs, effectiveness of path testing, etc.
- Predicates, path predicates and achievable paths, testing blindness and predicate expressions
- Path sensitizing – achievable and non-achievable paths, pragmatic observations and path instrumentation
- Implementation and application of path testing – new code, maintenance, re-hosting

4.1 INTRODUCTION

Path testing is the cornerstone of testing as it is based in the used of the program's control flow as a structural model. In this lesson we will discuss in detail the methods of generating tests from the program's control flow, criteria for selecting path, and how to determine path-forcing input values.

4.2 PATH TESTING BASICS

4.2.1 Motivation and Assumption

Path testing is the name given to the family of the testing techniques based on judiciously selecting a set of test paths through a program. Some measure of test thoroughness is achieved if the set of test paths is correctly chosen, e.g. pick up those paths which ensure that every statement has been executed at least once.

Path testing is one of the oldest techniques of structural testing. Every person who examines software testing in depth wants to cover each statement and branch at least once under some test. This technique was the first to come under theoretical scrutiny.

It is the most applicable to new software for unit testing. It is a structural technique which requires complete knowledge of the structure of the code. It is normally used by the programmers to unit test their own code. The effectiveness of this technique deteriorates as the size of the software increases. It is rarely used for system testing.

The Bug Assumption

- The bug assumption for path-testing strategies is that something has gone wrong with software that makes it take a different path than the desired, e.g. 'GOTO X' where 'GOTO Y' was intended.
- We assume that the specifications are correct and achievable, that there are no processing bugs other than those that affect the control flow, and that data is properly defined and accessed.

Structured programming languages prevent many of the bugs targeted by path testing for these languages are reduced. Consequently, old code has a higher proportion of control flow bugs than contemporary code and for such software path testing is indispensable e.g. Assembly languages like COBOL, Basic and FORTRAN.

4.2.2 Control Flow-graphs

The control flow-graph (or flow-graph) is a graphical representation of a program's control structure. It uses the elements as shown is Figure 4.1: process blocks, decisions and junctions. It is similar to the flowchart.

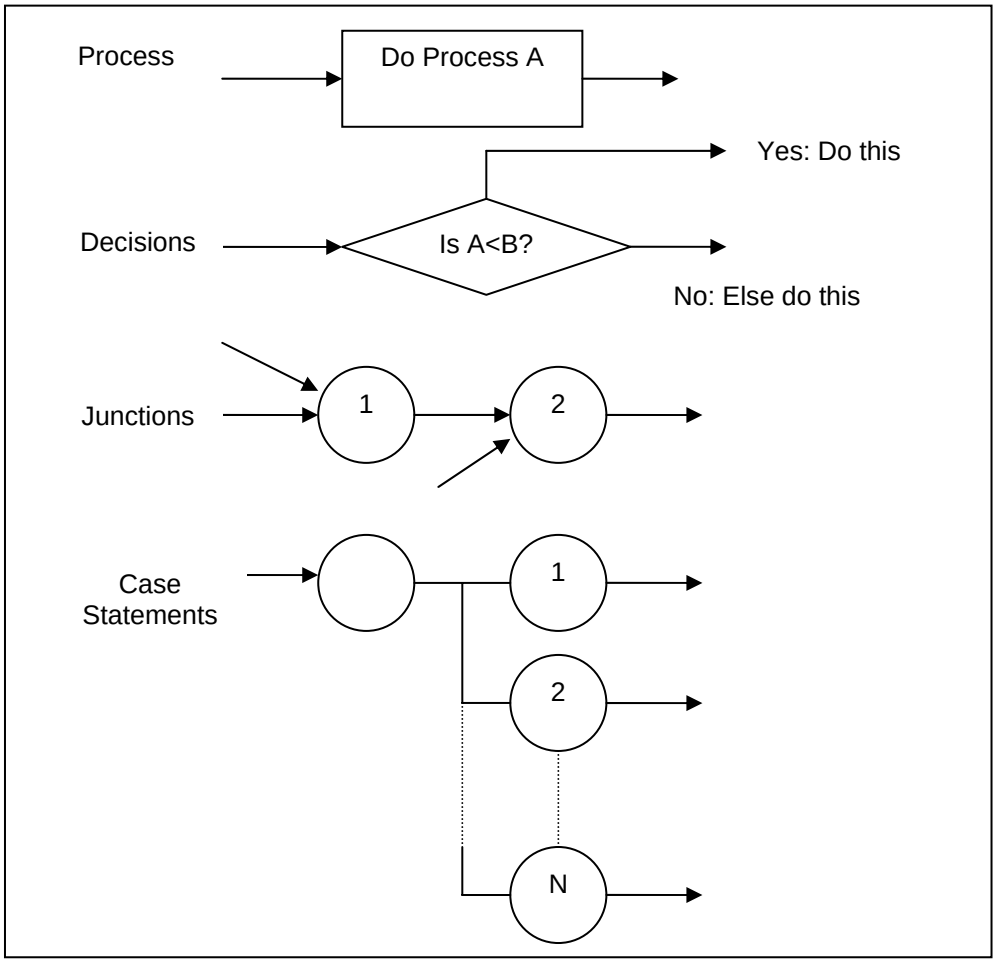


Figure 4.1: Flow-graph Elements

Process Block

A process block is a sequence of statements which do not contain any decision or junctions. Thus, in a process block, if one statement is executed, all the thereof will be executed. It can have one or hundreds of statements. Once a process is initiated, every statement contained within it will be executed and no statement within it will be a target for a GOTO statement.

A process has one entry and exit but one/multiple statement or instruction, a macro or a function call or a sequence of these. Thus, while designing test cases using a control flowgraph, it is not required to go into the details of the operations of the block. Even if a process affects the control flow, it will be manifested at the subsequent decision or case statement.

Decisions and Case Statements

A decision is a point in a program at which the control flow can diverge, e.g. conditional branch and skip statements. Majority of the decision statements are binary or two-way, but some are three way also. Test case designing for two-way conditions is easier than that of three-way conditions. A case statement is an example of a multiple way branch or decision.

Junctions

A junction is a point in the program where the control can merge, e.g. labels of a GOTO statement, END-IF and CONTINUE statements, etc.

Unconditional jumps or branches are not fundamental to programming. Although such statements are not essential to programming, their use can be avoided. Testing of such programs is also difficult as is the test designing.

Control Flowgraphs versus Flowcharts

The control flow graph resembles the flowchart of the program but differs in one way. We don't show details of what is their in a process block in a control flowgraph. A process-block is shown as a single block no matter the number of statements it comprises of internally. Conversely, in a flowchart, every statement of the process block is drawn. The flowchart focuses on process steps whereas the control flowgraph ignores these. The flowchart forces an expansion of visual complexity by adding multiple off-page connectors which confuse the control flow, but the flowgraph compresses the representation and makes it easier to flow.

Thus, the practice of drawing a flowchart may not be an effective design process but the act of drawing a flowgraph helps us to clarify the control flow and data flow related issues.

4.2.3 Notational Evolution

A control flowgraph is a simplified representation of a program's structure. We will take the example of a program (in Figure 4.2) to understand the same.

```
                    INPUT A, B
                    X = A + B
                    Y = A - B
                    IF X > 0 GOTO LABEL1
LABEL2: X = X – 1
LABEL1:X = X + Y
                    END
```

Figure 4.2: Program Example

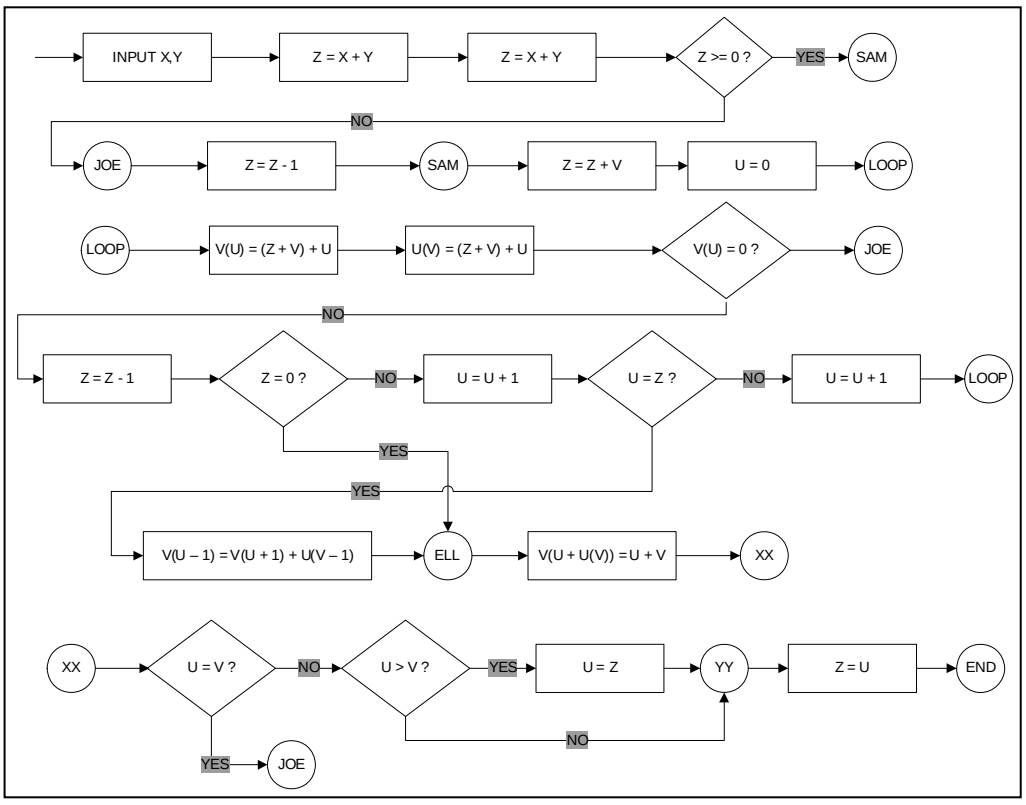


Figure 4.3: One-to-One Flowchart for Figure 4.2 Example

Figure 4.3 depicts the one-to-one for this program. Note that the complexity has increased and clarity has decreased.

In Figure 4.4, we have merged all the steps into a single process block. We now have a control flowgraph. But this notation is still complex and we have reduced it to achieve Figure 4.5. This is the actual representation of a control flowgraph. We have done a few notational changes in order to come up with this notation.

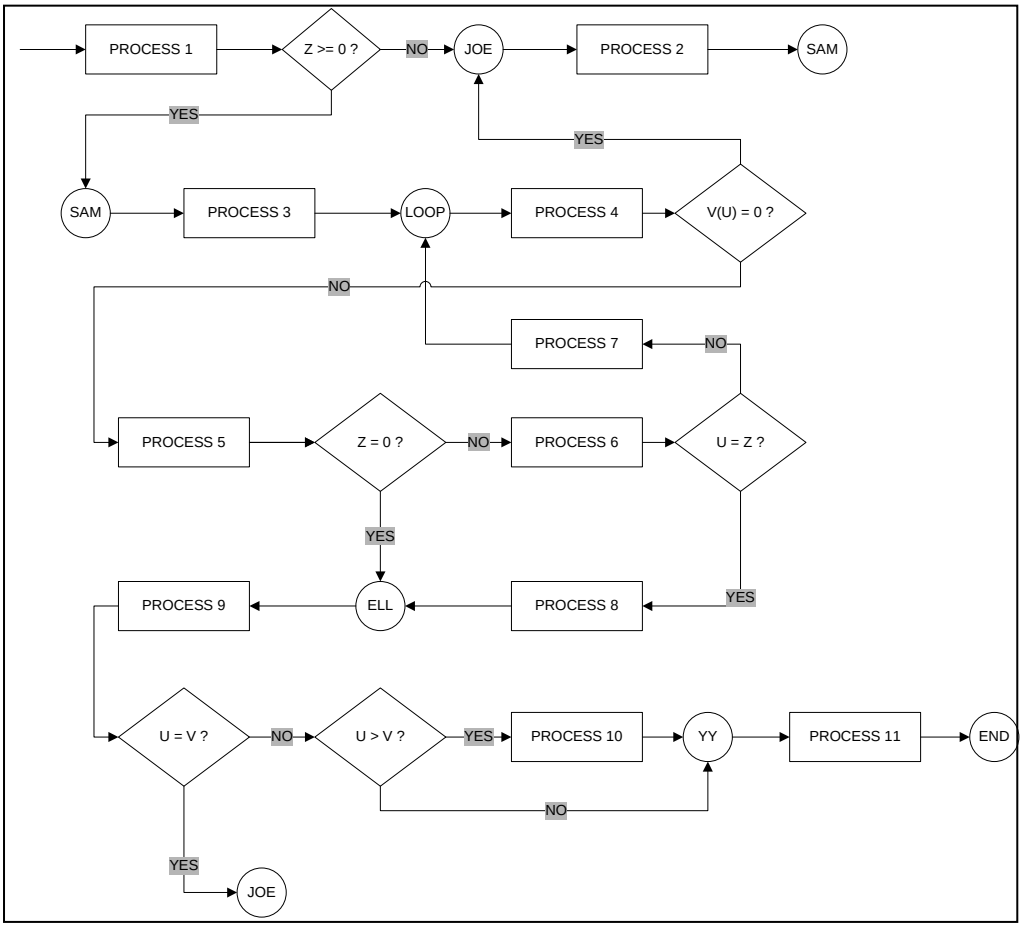


Figure 4.4: Control Flowgraph for Figure 4.3

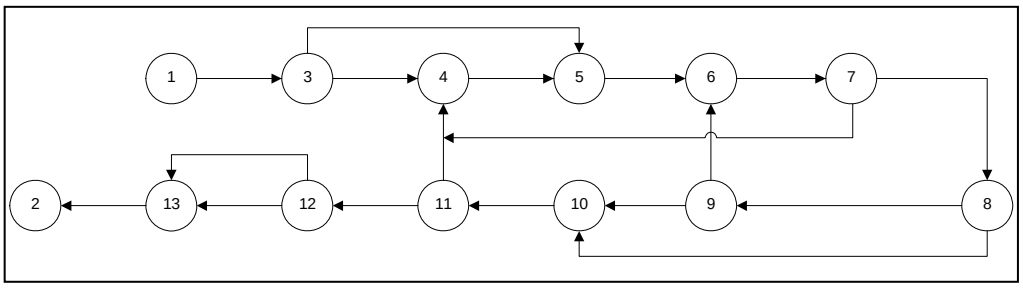


Figure 4.5: Simplified Flowgraph Notation

To do so we made several notational changes.

1. The process boxes aren't needed. There is an implied process at the junction of every line and decisions.
2. We need not know the specifics of the decisions, just the fact that there is a branch are sufficient.
3. The specific names of the target are not important but just the fact that they exist. So they can be replaced by simple numbers.

There are two kinds of components: circles and arrows joining these circles. A circle with more than one arrow leaving is a decision; a circle with more than one arrow entering is a junction. We call the circles nodes and the arrow links. Entry and exit are also considered as nodes. Numbering of the nodes is done as per the original program labels. The link name can be formed from the names of the nodes it spans, e.g. the link between node 4 and 7 is denoted as link (4, 7). An alternate to this technique is to use a lowercase character to denote each link.

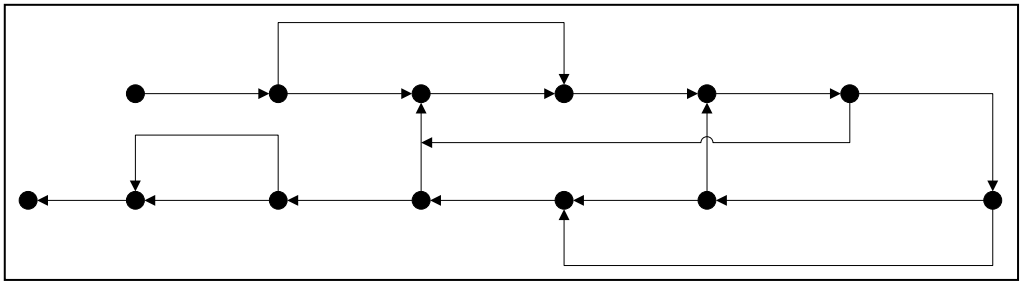


Figure 4.6: Even Simpler Flowgraph Notation

The final transformation is shown in Figure 4.6 where we have removed the node numbers to achieve an even simpler notation. This can be further reduced to the linked-list notation as shown in Figure 4.7.

```
1     (BEGIN)          :   3
2     (END)            :
3     (Z>0?)           :   4 (FALSE)
                       :   5 (TRUE)
4     (JOE)            :   5
5     (SAM)            :   6
6     (LOOP)           :   7
7     (V(U)=0?)        :   4 (TRUE)
                       :   8 (FALSE)
8     (Z=0?)           :   9 (FALSE)
                       :   10 (TRUE)
9     (U=Z?)           :   6 (FALSE) = LOOP
                       :   10 (TRUE) = ELL
10    (ELL)            :   11
11    (U=V?)           :   4 (TRUE) = JOE
                       :   12 (FALSE)
12    (U>V?)           :   13 (TRUE)
                       :   13 (FALSE)
13                     :   2 (END)
```

Figure 4.7: Linked-List Control Flow Graph Notation

4.2.4 Path Testing

Paths, Nodes and Links

A path is a sequence of instructions or statements that start at an entry, junction or decision and at another or same entry, junction or decision, or exit, through a program. A single process, junction or decision may be visited more than once in a path. Path consists of segments. The smallest segment is a link i.e. a single process that lies between two nodes. A path segment is a succession of consecutive links that belongs to some path. The length of a path is measured by the number of links in it or alternatively, by the number of nodes traversed. The latter method has some typical theoretical and analytical advantages. If a program is assumed to have a single entry and exit node, then the number of links is just one less than the number of nodes traversed. Because links are named by the pair of nodes joined by them, the name of a

path is the name of nodes along that path. For example, the shortest path in the Figure 4.5 from entry to exit is called "(1, 3, 5, 6, 7, 8, 10, 11, 12, 13, 2)". Alternatively, if we choose to label the links, the name of the path is the succession of link names along the path. A path has a loop if any node is repeated. For example, path (1, 3, 4, 5, 6, 7, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 2) in Figure 4.5 loops about nodes 4, 5, 6 and 7.

The word path is used in a restricted sense to denote a path that starts at a routine's entrance and ends at its exit. In reality, test paths are entry to exit paths. The term entry/exit paths and complete paths are used to denote the paths that start at an entry node and goes to an exit. We will study these in detail here because: (1) it's difficult to set up and execute paths that start at an arbitrary statement; (2) it's hard to stop at an arbitrary statement without setting traps or using patches; and (3) entry/exit paths are what we want to test because we use routines that way.

Multi-Entry/Multi-Exit Routines

If a routine performs several variations on the same processing and it is effective to bypass part of the processing, the correct way to design the routine is to provide an entry parameter that within the routine, directs the control flow to the proper point. Similarly, if a routine can have several different kinds of outcomes, then an exit parameter should be used. An alternative could be to encapsulate the common parts into subroutines. Instead of using direct linkages between multiple exits and entries, we handle the control flow by examining the values of the exit parameter that can serve as an entry parameter for the next routine or a return parameter for the calling routine.

The trouble with multi-entry and multi-exit routine is that it can be very difficult to determine what the inter-process control flow is, and consequently it is easy to miss important test cases. Further the use of such routines increases the number of entries and exits and thus, the number of interfaces, which means more test-cases than otherwise required.

Fundamental Path Selection Criteria

There exist multiple paths between the entry and exit of a typical routine. The path doubles at every decision and gets multiplied with the number of iterations at every loop. Each pass through a loop constitutes a separate path. Now, in such a case, how can we ensure "complete testing" of all the paths?

1. Exercise every path from entry to exit.
2. Exercise every statement or instruction at least once.
3. Exercise every branch and case statement, in each direction, at least once.

If prescription 1 is followed, automatically 2 and 3 are also followed. But prescription 1 is impractical for most of the routines. It can be done only for those routines that have no loops. A static analysis of the code cannot determine whether a piece of code is reachable or not. A dynamic analysis can only determine whether a code is reachable or not and can thus distinguish between the ideal structure we think we have and the actual, buggy structure.

Path Testing Criteria

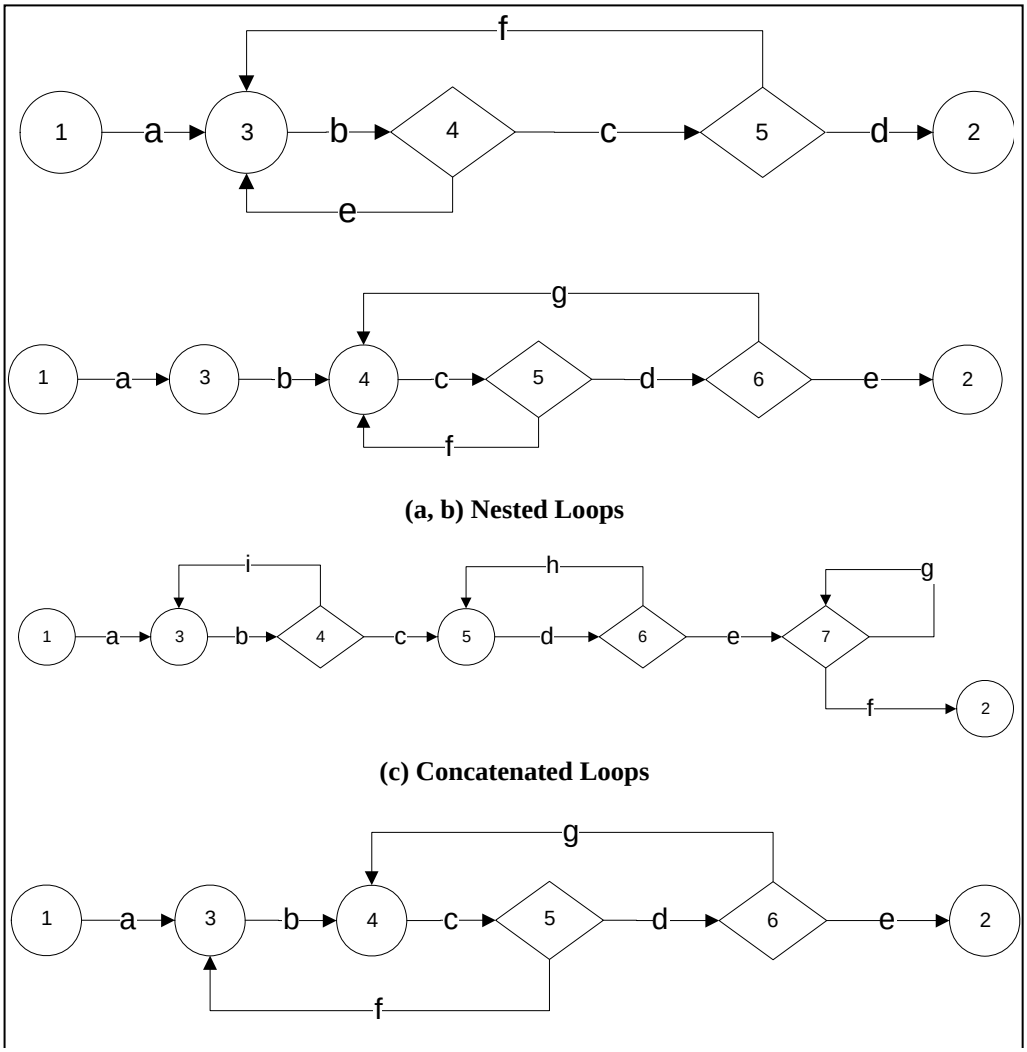
Any testing strategy based on paths must at least exercise every instruction and take branches in all directions. A set of test that does this is complete, not in absolute sense, but in the sense that anything less must have left something untested. Thus, we have explored three different testing criteria or strategies out of a potentially infinite family of strategies. These are:

1. ***Path testing* (P∞)*:*** Execute all possible control flow paths through the program; typically all possible entry/exit paths through the program. If we can achieve this, we are said to have completed 100% path coverage. This is the strongest criterion in the path testing strategy family and generally impossible to achieve.
2. ***Statement Testing* (P1)*:*** Execute all statements in the program at least once under some test. If we do enough tests to achieve this, we are said to have achieved 100% statement coverage or alternatively, 100% node coverage. We denote this by C1. This is the weakest criterion in the family; testing less than this is unconscionable and should be criminalized.
3. ***Branch Testing* (P2)*:*** Execute enough tests to assure that every branch alternative has been exercised at least once under some test. If we do enough tests to achieve this prescription then we have achieved 100% branch coverage or alternatively 100% link coverage. For structure software, branch testing and thus branch coverage strictly includes statement coverage. We denote branch coverage by C2.

We can define innumerable such definitions but none of them is sufficient to carry out exhaustive testing.

4.2.5 Loops

There are three kinds of loops: nested, concatenated and horrible loops, as shown in Figure 4.8.



(a, b) Nested Loops

(c) Concatenated Loops

Contd...

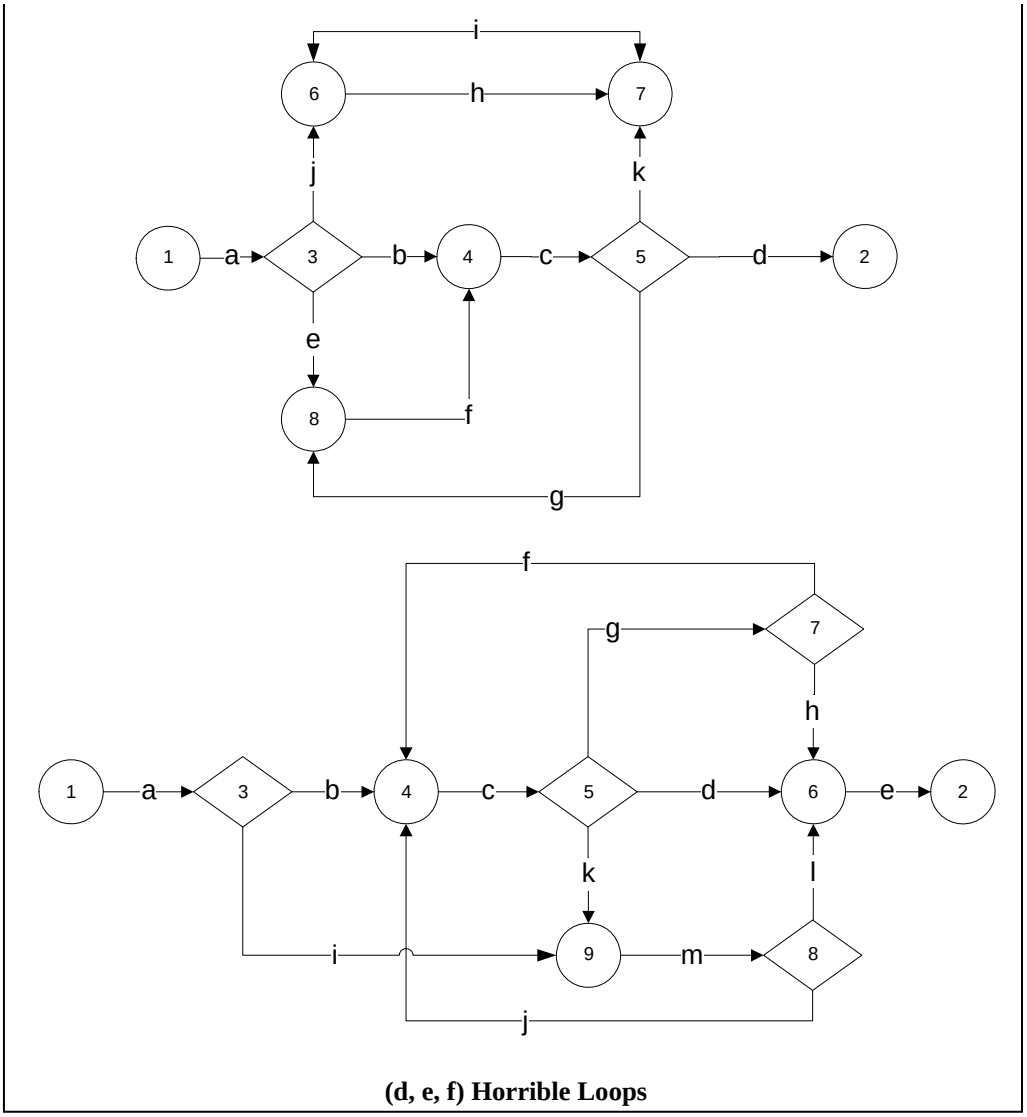


Figure 4.8: Examples of Loop Types

Nested Loops

We can reduce the burden of testing nested loops using the following tactics:

1. Start at the innermost loop and set all the outer loops to their minimum values.
2. Test the minimum, minimum + 1, typical, maximum – 1, and maximum for the innermost loop, while holding the outer loops at their minimum-iteration-parameter values. Tests can be expanded for out-of-range and excluded values.
3. After doing the outermost loop, GOTO step 5, ELSE move out one loop and set it up as in step 2 – with all other loops set to typical values.
4. Continue outward in this manner until all the loops have been covered.
5. Do the five cases for all loops in the nest simultaneously.

Possible bug could be an unbounded processing time. Further expansion of these cases is possible by accounting for the problems associated with initialization of variables and with excluded combinations and ranges.

Concatenated Loops

These loops fall between single and nested loops in terms of test cases. Two loops are said to be concatenated if it is possible to reach one after exiting the other while on the same path from entrance to exit. If they are on different paths then they are not concatenated but two individual loops. Loops on the same path, but independent can be treated as individuals; but if the iteration values in one loop are directly or indirectly related to the iteration values of another loop, and they can occur on same path, then they can be treated as nested loops. The problem of excessive processing time for combination of loop-iteration values should not occur because the loop-iteration values are additive and not multiplicative as with the nested loops.

Horrible Loops

Although there are some techniques to test such loops, but it is always advisable to avoid using them due to their use of code that jumps into and out of loops, hidden loops, and cross-connected loops, makes iteration-value selection for test cases an awesome and ugly task.

Loop Testing Time

Testing of any loop can consume a lot of time, especially if all the possible extreme values are to be attempted. It gets worse for nested and dependent concatenated loops. In order to test nested loops in which testing the combination of extreme values leads to long test times, we have several options:

1. Exhibit that the combined execution time results from an unreasonable specification and fix that specification.
2. Prove that although the combined extreme cases are hypothetically possible, they are impossible in the real world i.e. they cannot occur.
3. Put in limits or checks that prevent the combined extreme cases and test the software that implements such limits.
4. Test with extreme combinations, but using different numbers.

Check Your Progress 1

1. State whether the following statements are true or false:
   (a) Path testing is rarely used for system testing.
   (b) The control flow-graph ignores the process steps.
   (c) Possible bug of a nested loop could be an unbounded processing time.
2. Give the two assumptions for bugs during path testing.

   .....

   .....

4.2.6 Variations

Branch and statement testing are easy to implement, effective and reasonable. But for the case of the complicated tests in the path testing family, we can use two main classes of variations:

1. Strategies between P2 and total path testing.
2. Strategies weaker than P1 or P2.

Stronger strategies require more complicated path selection criteria, majority of which are impractical for human test design. Typically, strategy is embedded in a tool which selects a covering set of paths based on strategy or helps the programmer in doing so.

The weaker strategies based on doing less than C1 or C2 seem to contradict our position that C2 is the minimum requirement. This is true for completely new software but not for those in the maintenance situation.

4.3 PREDICATES, PATH PREDICATES AND ACHIEVABLE PATHS

4.3.1 Predicates

The value of the decision variable decides the direction taken at the decision. For binary decisions, decision is taken in the form of a Boolean expression whose outcome is either true or false. The logical function evaluated at a decision is called a predicate. Every path corresponds to a succession of TRUE/FALSE values for the predicates traversed on that path. A predicate associated with a path is called a path predicate.

Multiway Branches

The path taken through a multiway branch such as GOTO, case statements, or jump statements (in assembly language) cannot be directly expressed in the terms of TRUE/FALSE. Although such alternatives can be described in the terms of multivalued logic, an easier expedient is to express multiway branches as an equivalent set of IF..THEN..ELSE statements. This translation may not be unique as there are several ways create a tree of IF..THEN..ELSE.

Inputs

In testing, input does not only refer to the direct inputs in the form of variables in a subroutine call, but includes all data objects referenced by the routine whose values are fixed prior to entering it, e.g. inputs in a calling sequence, objects in a data structure, values left in a register. Inputs can be treated as numbers irrespective if they are numbers, Boolean, integers, etc. Because an array can be mapped onto a one-dimensional array, we can treat the set of inputs to the routine as if it is a one-dimensional array, which we call input vector.

4.3.2 Predicate Expressions

Predicate Interpretation

The act of symbolic substitution of operations along the path in order to express the predicate solely in terms of the input vector is called predicate expression. The interpretation may depend upon the path. For example,

```
INPUT X
ON X GOTO A,B,C,….
A: Z := 7 @ GOTO HEM
B: Z := -7 @ GOTO HEM
C: Z := 0 @ GOTO HEM
…..
HEM: DO SOMETHING
```

….

HEN: IF Y + Z > 0 GOTO ELL ELSE GOTO EMM

The predicate interpretation at HEN depends upon the path we took through the first multiway branch. It includes three cases i.e. “IF Y + 7 > 0 GOTO…”, “IF Y – 7 > 0 GOTO…” and “IF Y > 0 GOTO…”. Because every path can lead to a different interpretation of predicates along that path, a predicate that after interpretation does not depend on input values does not necessarily constitute a bug.

The path predicates are the specific form of the predicates of the decisions along the selected path after interpretation.

Independence and Correlation of Variables and Predicates

The path predicates take on truth values (TRUE or FALSE) based on the values of input variables, either directly or indirectly. A variable is independent of processing if its value does not change as a result of processing. Conversely, the variable is said to be process dependent if its value changes as a result of processing. Similarly, a predicate whose truth value changes as a result of processing is said to be process dependent and whose value does not change as result of the processing is process independent. Process dependence of a predicate does not necessarily mean that it depends upon the input variables on which it is based. However, if all the variables on which a predicate is based are process independent, it follows that the predicate must be process independent and therefore its truth value is determined by the inputs directly.

Variables, whether process dependent or independent, may be correlated to one another. Two variables are correlated if every combination of their values cannot be independently specified. Variables whose values can be specified independently without restriction are uncorrelated. By analogy, a pair of predicates whose outcome depend on one or more variables in common (whether or not those variables are correlated) are said to be correlated predicates.

4.3.3 Predicate Coverage

Compound predicates are those of the form A .OR. B or A .AND. B and more complicate Boolean expressions. The branch taken at such decisions is determined by the truth value of the entire Boolean expression.

Predicate coverage is said to have been achieved if all the possible combinations of truth values corresponding to the selected path have been explored under some test. It is stronger than branch coverage. If all the possible combinations of all predicates are covered under all interpretations then it is equivalent to total path testing.

4.3.4 Testing Blindness

Testing blindness is a pathological situation in which the desired path is achieved for wrong reason. It can occur because of the interaction of two or more statements that make the buggy predicate work in spite of its bug and because of an unfortunate selection of input values that does not reveal the situation. We will be discussing three kinds of testing blindness: Assignment blindness, equality blindness and self-blindness.

Assignment Blindness

Assignment blindness occurs when the erroneous predicate appears to work correctly because the chosen value works with both correct and incorrect predicate assignment statement.

For example,

Correct	**Incorrect**
X := 7	X := 7
…..	…..
IF Y > 0 THEN	IF X + Y > 0 THEN

If a test applies Y := 1, the desired path is taken in either case, but there still exists a bug if a different value is assigned to X. In this case, the wrong path would be taken because of the error in the predicate.

Equality Blindness

Equality blindness occurs when the path selected by a prior predicate results in a value that works both for correct and erroneous predicate. For example,

Correct	**Incorrect**
IF Y := 2 THEN …	IF Y := 2 THEN …
…..	….
IF X + Y > 3 THEN…	IF X > 1 THEN …

The first predicate forces the rest of the path so that for any positive value of X, the path taken at the second predicate will be the same for the correct and erroneous versions.

Self-blindness

Self-blindness occurs when the erroneous predicate is a multiple of the correct predicate and as a result is indistinguishable along that path. For example,

Correct	**Incorrect**
X := A	X := A
…..	….
IF X – 1 > 0 THEN …	IF X + A – 2 > 0 THEN …

The assignment X = A, makes the two predicates multiples of each other, i.e. A – 1 > 0 and 2A – 2 > 0, so the direction taken is the same for the correct and incorrect version. A path with another assignment could behave differently and would expose the bug.

4.4 PATH SENSITIZING

Let us first review the progression of thinking to this point.

1. We want to select and test enough paths to achieve a satisfactory notion of test completeness C1 and C2.
2. Extract the program's flow control flowgraph and select a set of tentative covering paths.
3. For any path in that test, interpret the predicates along the path as needed to express them in terms of the input vector. In general, individual predicates are compound or may become compound as a result of interpretation.
4. Trace the path through, multiplying (Boolean) the individual compound predicates to achieve a Boolean expression such as

   (A + BC)(D + E)(FGH)(IJ)(K)(L)

Where terms in the parentheses are the compound predicates met at each decision along the path and each letter (A, B, ….) stands for simple predicates.

5. Multiply out the expressions to achieve a sum-of-products form:

   ADFGHIJKL + AEFGHIJKL + BCDFGHIJKL + BCEFGHIJKL

6. Each product denotes a set of inequalities that, if solved, will yield an input vector that will drive the routine along the designated path.
7. Solve any one of the following sets for the chosen path and you have found a set of input values for the path.

If you can find the solution, then the path is achievable. If you can't find a solution to any of the sets of inequalities, the path is unachievable. The act of finding a set of solutions to the path predicate expression is called path sensitization.

Heuristic Procedures for Sensitizing Path

Rather than selecting paths which we don't know how to sensitize, we try to choose a covering path set that is easy to sensitize. We can choose the hard to sensitize paths only to ensure complete coverage.

1. Identify all the variables that affect the decisions and classify them as either process dependent or independent. Also, identify correlated input variables. For dependent variables, express the nature of the process dependency as an equation, function or whatever is convenient and clear. For correlated variables, express the logical, arithmetic or functional relation that defines the correlation.
2. Classify the predicates as dependent or independent. Identify correlated input predicates and document the nature of the correlation as for variables. If the same predicate appears at more than one decision, the decisions are obviously correlated.
3. Start path selection with uncorrelated, independent predicates. Cover as much as you can. If you achieve coverage and you had identified supposedly dependent predicates, something is wrong. i.e. Your tracing path could be faulty or the predicates are incorrectly classified or there is a bug or the predicates are corrected and/or dependent in such as way so as to nullify the dependency.
4. If coverage hasn't been achieved using independent uncorrelated predicates, extend the path set by using correlated predicates; preferably those whose resolution is independent of the processing.
5. If coverage hasn't been achieved, extend the cases to those that involve dependent predicates (typically required to cover loops), preferably those that are not correlated.
6. Last, use correlated, dependent predicates.
7. For each path selected above, list the input variables corresponding to the predicates required to force the path. List the value of the variable if it is independent. List the relation of the variable that will make it go the right way, if it is dependent. State the nature of the correlation of the variable if it is correlated.
8. Each path will result in a set of inequalities which must be simultaneously satisfied to force the path.

Check Your Progress 2

Fill in the blanks:

1. ..... occurs when the erroneous predicate is a multiple of the correct predicate and as a result is indistinguishable along that path.
2. ..... occurs when the erroneous predicate appears to work correctly because the chosen value works with both correct and incorrect predicate assignment statement.
3. A OR B, A AND B, etc are examples of ..... predicates.
4. A variable is ..... if its value does not change as a result of processing.

4.5 PATH INSTRUMENTATION

4.5.1 Problem

The outcome of a test is what we expect to happen as a result of testing. This includes the outputs as well. Just like inputs, the test outcomes include computer memory, mass storage, I/O, registers that should or should not have changed as a result of tests conducted. When a test is run, we compare the actual outcome with the expected outcome. If these match, we can say that some of the conditions for passing have been satisfied but these are not sufficient as the desired outcome could have been achieved for the wrong reason. This situation is called coincidental correctness. Similarly, let us assume that we ran a covering of tests and achieved the desired outcomes for each case. Again, we cannot say that we have covered because the desired outcome could have been achieved from the wrong path. Path instrumentation is what is required to be done to confirm that the outcome was achieved by the intended path.

All instrumentation methods are the variations of an interpretive trace. An interpretive trace program is the one that executes every statement in order and records the intermediate values of all calculations, the statements labels traversed, etc. The problem with these traces is that they give us far more information than what is needed. Looking at the limitations of the trace packages or symbolic debuggers, a variety of instrumentation methods have emerged which are more suitable for testing.

4.5.2 Link Markers

A simple and effective instrumentation is called a traversal marker or link marker. Every link is named by a letter. Instrument the links so that its name is recorded when it is executed. The series of letters produced while going from the routine's entry to its exit should exactly correspond to the path name if there are no bugs. Also, a single link marker may not be sufficient because links can be affected due to bugs.

4.5.3 Link Counters

The method based on counters is less disruptive. Instead of pushing unique link names while traversing the link, we simply increment the counter. The same problem, as with single link markers, exists with single link counters. This leads to double link counters. With these, we expect an even count which is double the expected path length.

4.5.4 Implementation

For the source languages that support test tool, path instrumentation and verification can be provided using this tool for unit testing. But for the languages that do not support these tools, it has to be done the hard way.

The introduction of probes may lead to bugs especially when they are inserted by hand. Automatically inserted probes are less prone to error but can be inserted in terms of real and not intended structure. This discrepancy is greater if the control is affected by what goes in low level routines that are called by the routines under test. Instrumentation is more important when the path testing is used at higher levels of program structure e.g. in transaction flows rather than in unit level. Also, the discrepancy between actual and intended structure becomes greater at the higher level of structure, however, the instrumentation overhead is smaller.

Probe installation is easier when the programming language supports conditional assembly or conditional compilation. The probes are built into source code and tagged into categories. Both counters and traversal markers can be implemented as only a part of probes would be assembled at a time. Very rarely will all the probes be compiled and activated.

Conditional assembly or compilation must be used with caution especially at the higher program levels. A unit may take just a few seconds to compile or assemble, but, the same unit if compiled in the context of a complete system, could take hours, thereby nullifying many of the advantages of conditional assembly or compilation.

You can also use macros or function calls if conditional assembly or compilations are not available. The probe can be turned on or off by changing the macro or function definition or be using ON/OFF parameters inside the functions or macros.

4.6 INSTRUMENTATION AND APPLICATION OF PATH TESTING

4.6.1 Integration, Coverage and Paths in Called Components

Path testing methods are used in unit testing especially for new software. The new component is tested as an independent unit with all called components and corresponding components replaced by stubs. This is the conventional unit testing method. Path-testing method at this stage is used to deal with the potential control-flow problems without the distraction of possible bugs in called co-requisite components. We integrate the components carefully probing the interface issues. Once the interfaces are tested, we re-test the integrated component replacing the stub with real subroutines and co-requisite components. The component is now ready for the next level of integration. This bottom-up integration will continue till the entire system has been integrated.

Integration proceeds in associated blocks of components while a bottom-up integration strategy may be used in parts. Stubs may be correctly avoided because its bug potential may be higher than that of a real routine.

Path testing relies on the assumption that we can do effective testing one level at a time without being overly concerned with what happens at the lower levels.

We typically loose around 15% coverage with each level. Thus, while we may achieve C2 at the current level, path tests will achieve 85% one level down, 70% two levels down, and so on. When the testing by all methods has been considered, C1 coverage at the system level ranges from 50% to as high as 85%. No statistics are available for C2 coverage in the system testing as it is impossible to monitor C2 coverage without disrupting the system’s operation to the point where testing is impossible. System level coverage is restricted to C1, which can be done by tools that minimally disturb the system.

4.6.2 New Code

Completely new or substantially modified code should be subjected to enough path testing to achieve C2. Stubs are used where it is clear that the bug potential of the stub is lower than that of the called component. Thus, old and trusted components will not be replaced by stubs. Paths within called components are given consideration but only to that extent that the path chosen is achievable at a higher level. The unit test suite must be automated so that it can be repeated as integration progresses. It allows redoing most of the tests with very little effort as we achieve larger aggregated components. A previously selected path is unachievable means that we have found a bug from an unsuspected interaction.

4.6.3 Maintenance

The situation in the case of maintenance is different. Path testing would be used first for the modified components but called and co-requisite components will be invariably real rather than simulated. If we have a configuration-controlled, automated, unit test suite, then path testing will be repeated entirely with such modifications as required to accommodate the changes. Otherwise, selected paths will be chosen in an attempt to achieve C2 over the changed code.

4.6.4 Re-hosting

Path testing with the coverage C1 + C2 is a powerful tool for re-hosting old software. When it is used in conjunction with automatic or semiautomatic structural test generators, we get an effective re-hosting process. When software is no longer cost-effective to support the environment in which it runs, it is re-hosted. The objective of re-hosting is to change the environment and not the software.

We can do this in the following manner. First, a translator from the old to the new language is created and tested. The bugs in the re-hosted software will be in the translating algorithm and the translator, if any. The re-hosting process is intended to catch such bugs. Second, a complete (C1 + C2) path test suite is created for the old software in the old environment. The suite is run on the old environment on the old software and all the outcomes are recorded. These ten become the specification of the re-hosted software. Test failures or incomplete coverage leads to changes in the translators that could necessitate translation reruns and retesting.

The cost of the process is comparable to cost of rewriting the software from scratch; it could be even more expensive, but that’s not the point. This method avoids the risks associated with rewriting and achieves a stable, correct software base in the new environment without operational or security compromises.

4.7 LET US SUM UP

Path testing based on structure is a powerful unit-testing tool. With suitable interpretation, it can be used for functional tests. The objective of path testing is to execute enough tests to assure that, as a minimum, C1+ C2 have been achieved. Select paths as deviations from the normal paths, starting with the simplest, most familiar, most direct paths from the entry to exit. Add paths as needed to achieve coverage.

Add paths to cover extreme cases for loops and combinations of loops: no looping, once, twice, one less than maximum, the maximum. Attempt forbidden cases. Find path-sensitizing input data sets for each selected path. If a path is unachievable, choose another path that will also achieve coverage. Use instrumentation and tools to verify the path and to monitor coverage.

Incorporate the notion of coverage (Especially C2) into all reviews and inspections. Make the ability to achieve C2 a major review agenda item. Design test cases and path from the design flow-graph or PDL specification but sensitize paths from the code as part of desk checking. Do covering test case designs either prior or concurrently with coding. A test that has revealed a bug is successful and not a failure.

4.8 LESSON END ACTIVITIES

1. Differentiate between motivation and assumption behind software path testing.
2. What are predicates, path predicates and achievable paths? Explain the relation between all of them with respect to software testing.
3. What are the various methods used for path instrumentation? Explain each of them.

4.9 KEYWORDS

Control flow-graph: Control flow-graph (or flow-graph) is a graphical representation of a program's control structure.

Process block: A process block is a sequence of statements which do not contain any decision or junctions.

Decision: A decision is a point in a program at which the control flow can diverge.

Junction: A junction is a point in the program where the control can merge.

Predicate: The logical function evaluated at a decision is called a predicate.

Path sensitization: The act of finding a set of solutions to the path predicate expression is called path sensitization.

Predicate expression: The act of symbolic substitution of operations along the path in order to express the predicate solely in terms of the input vector is called predicate expression.

4.10 QUESTIONS FOR DISCUSSION

1. Describe the various applications of path testing.
2. Write short notes on:
   (a) Control flow-graphs
   (b) Loops
   (c) Path testing
   (d) Variations
3. What are achievable and non-achievable paths? Elaborate on these.

Check Your Progress: Model Answers

CYP 1

1. (a) True
   (b) True
   (c) True

Contd….

2. Something has gone wrong with software that makes it take a different path than the desired. The specifications are correct and achievable, that there are no processing bugs other than those that affect the control flow, and that data is properly defined and accessed.

CYP 2

1. Self-blindness
2. Assignment blindness
3. compound
4. independent of processing

4.11 SUGGESTED READINGS

Boris Beizer, *Software Testing Techniques*, Second Edition, Dreamtech Press, 2003.

Myers and Glenford, J., *The Art of Software Testing*, John-Wiley & Sons, 1979.

Roger, S. Pressman, *Software Engineering – A Practitioner's Approach*, 5th Edition, Mc-Graw Hill, 2001.

Marnie, L. Hutcheson, *Software Testing Fundamentals*, Wiley-India, 2007.

UNIT III

LESSON

5

TRANSACTION FLOW TESTING

CONTENTS

5.0 Aims and Objectives

5.1 Introduction

5.2 Transaction Flows

- 5.2.1 Usage
- 5.2.2 Implementation
- 5.2.3 Complications
- 5.2.4 Transaction-flow Structure

5.3 Transaction Flow Testing Techniques

- 5.3.1 Inspections, Reviews and Walkthroughs
- 5.3.2 Path Selection
- 5.3.3 Sensitization
- 5.3.4 Instrumentation
- 5.3.5 Test Databases
- 5.3.6 Execution

5.4 Implementation Comments

- 5.4.1 Transaction-based Systems
- 5.4.2 Hidden Languages

5.5 Let us Sum up

5.6 Lesson End Activities

5.7 Keywords

5.8 Questions for Discussion

5.9 Suggested Readings

5.0 AIMS AND OBJECTIVES

After studying this lesson, you would be able to understand:

- Transaction flows: usage, implementation, complications
- Techniques of transaction-flow testing – inspections, reviews, walkthroughs, path selection, sensitization, instrumentation, etc.
- Transaction-based systems and hidden languages

5.1 INTRODUCTION

The control flow-graph is an infinite model based on links and nodes. It was introduced as a structural model. We are now using a different flow-graph based on the same concepts, the transactional flow graph. This is based on the behavioral model of the program that leads to functional testing. A functional focus leads us to functional test methods. This also involves representing the problem as a graph model and defining paths to test this model completely.

5.2 TRANSACTION FLOWS

A transaction is a unit of work from a system's point of view. It consists of a sequence of operations part of which is performed by a system, person, or devices out of system. Transactions begin with a birth i.e. they are created with an external act.

5.2.1 Usage

Transaction flows are must for specifying requirements of complicated systems. The transaction flows of big systems can go up to thousands e.g. airline reservation, air-traffic control, etc. These flows are represented using flow-graphs, many of which have a single path. Loops are infrequent compared to flow-graphs. The most common loop is to request user to re-enter his inputs after an error occurs.

5.2.2 Implementation

The implementation of a transaction flow is usually implicit in the design of a system's control structure and associated database. For example, there is no direct one-to-one correspondence between the "processes" and "decisions". A transaction flow is a representation of a path taken by a transaction through a succession of processing modules. We can think of each transaction as a token like a transaction-control block which is passed from routine to routine as it progresses through its flow. The transaction flow-graph is a pictorial representation of what happens to the tokens; it is not the control structure of the program that manipulates those tokens.

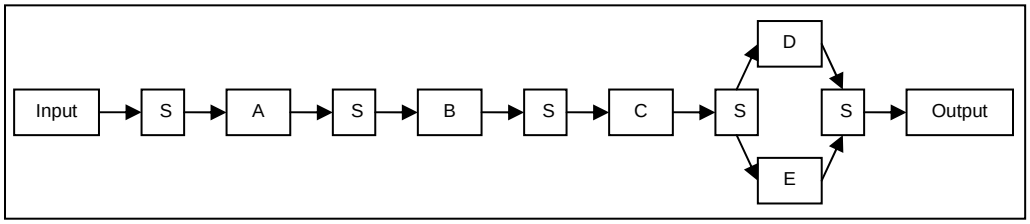


Figure 5.1: Transaction Flow

Figure 5.1 shows a transaction flow and the corresponding implementation of a program that creates that flow. This transaction goes through input processing, which classifies it as to type, and then passes through process A, followed by B. The result of process B may force the transaction to pass back to process A. The transaction then goes to process C, then to either D or E, and finally to output processing.

5.2.3 Complications

Although transactions have a unique identity in most of the cases right from the time they are created to the time they are completed, in many systems a transaction can give birth to others, and transactions can also merge. The simple flow-graph is inadequate to represent the transaction flows that split and merge.

Birth

Figure 5.2 shows three different possible interpretations of the decision symbol or nodes with two or more outlinks. Figure 5.2(a) shows a decision node, as in control flow-graphs, this symbol means that the transaction will either take one alternative or the other, but not both. It is a decision point in a transaction flow. Figure 5.2(b) shows a different situation wherein the incoming transaction (the parent) gives birth to a new transaction (the daughter), from where the two continue on their own paths, the parent retaining its identity as a transaction. This situation is called biosis. Figure 5.2(c) is similar to Figure 5.2(b) except that the parent transaction is destroyed and the two new transactions are created. This situation is called mitosis.

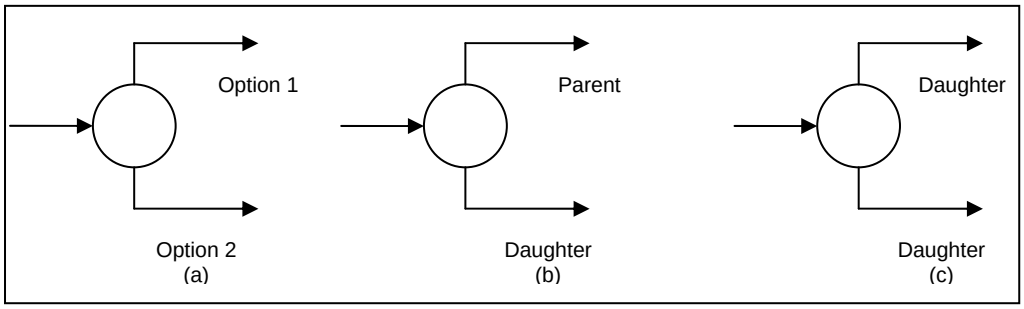


Figure 5.2: Nodes with Multiple Outlinks

Mergers

Transaction-flow junction points are as difficult as transaction-flow splits. Figure 5.3(a) shows the ordinary junction similar to the one in control flow-graph. The transaction can arrive at any of the two given links (link 1 or link 2). In Figure 5.3(b) (absorption) a predator transactions absorbs a prey. The prey is gone but the predator retains its identity. Figure 5.3(c) shows a different transaction in which the two parent transactions merge to form a new child. This method is known as conjugation.

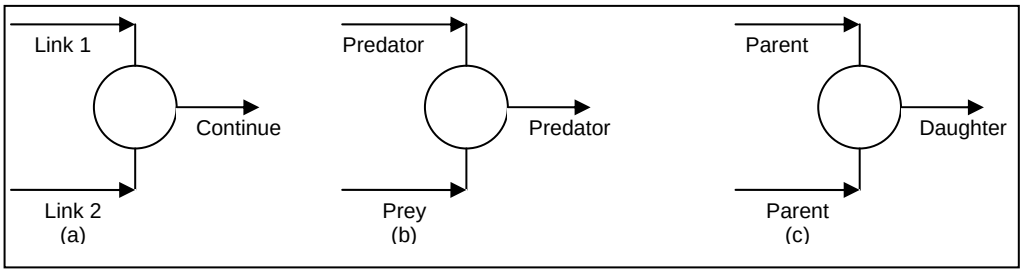


Figure 5.3: Transaction-flow Junctions and Mergers

5.2.4 Transaction-flow Structure

Transactional flows are normally ill structured in comparison with the control flow structures. The reasons behind this being:

1. It is model of both process and code.
2. Parts of the flow can incorporate the behavior of other systems over which there is no control.
3. There are no small parts to the totality of the transaction flows exists to model error conditions, failures, malfunctions and subsequent recovery actions. These are inherently unstructured – jumping out of loops, rampant GOTOs, etc.
4. The number of transactions and their complexities grow over time as more features are added and enhanced.

5. Systems are built on modules and the transaction flows result from the interaction of those modules. Good design is the one which does not involve changing modules to implement new transactions or modify the existing ones.

Check Your Progress 1

1. State whether the following statements are true or false:
   (a) Transactional flows are more structured in comparison with the control flow structures.
   (b) Transaction flow models comprise of both process and code.
2. Define transaction flows.

   .....

   .....

5.3 TRANSACTION FLOW TESTING TECHNIQUES

Complicated systems carrying out varied type of processing, complicated transactions must have explicitly represented transaction flows or the similar, documented. Equivalent representations like HIPO charts and Petri nets can also serve the same purpose. Transaction flows can also be represented in the form of PDLs which can be mapped to programs later on. They must clearly depict the transactions so that we can clearly obtain transaction flows from them. Transaction flows depict a lot of details.

Thus, we can conclude that in order to carry out transaction testing, it is necessary to get transactional flows. It is toughest and the most important step.

5.3.1 Inspections, Reviews and Walkthroughs

Transactional flow walkthroughs must begin at a preliminary stage of design as the designers must know what the system is supposed to do rather than how to implement that functionality. Let us discuss the review techniques in detail:

1. While conducting walkthroughs, we should:
   a. Discuss various transaction types which account for around 98% - 99% of the transactions the system has to process. We need to adjust the time spent and the intensity of the review proportional to the risks perceived. The designers must name the transactions, provide its flow-graph, identify all the processes, branches, loops, splits, mergers, etc.
   b. Discuss paths through flows in functional and not in technical terms. This discussion must be design independent.
   c. The designers must relate every flow to the specification and to show how that transaction follows the requirements, directly or indirectly. One-to-one correspondence may not be necessary because it may lead to poor implementation.
2. Transaction flow testing must be made the cornerstone of functional testing of the system. We need to carry out enough tests to ensure C1 and C2 coverage of the complete transaction flow-graphs.
3. Select extra transactional flow-graphs beyond C1 and C2 including loops, extreme values and domain boundaries.

4. Select additional transaction-flow paths for weird can very long and risky problems.

5.3.2 Path Selection

Path selection for system testing based on transaction flow graphs is quite different from the unit testing using the control flow graphs. We begin the covering tests (C1+C2) based on functionally sensible transactions as in the case of control flow-graphs. We begin with easier paths and go on to the tougher ones which in turn will expose missing interlocks, duplicated interlocks, interface problems, duplicated processing i.e. a lot things which otherwise would have shown up during the final acceptance testing or even after the system was operational.

5.3.3 Sensitization

Most of the normal paths are easy to sensitize – 80%-95% transaction flow coverage (C1 + C2) is usually easy to achieve. However, the remaining ones are difficult to achieve. Simple paths are easy to sensitize so much so that just identifying the normal path is enough to sensitize it. However, to sensitize the strange paths, you can use the following:

1. ***Use patches:*** It is a lot easier to fake an error return from another system by a judicious patch than it is to negotiate a joint test session. If we don't put a patch into our system, the interfacing system will have to put one into their system.
2. ***Mistune:*** Test in a system sized with resources which are 5%-10% of what one might expect to need. This helps to force most of the resource related exception conditions.
3. ***Break the Rules:*** Almost all transactions require associated and correctly specified data structures to support them. These are created using a system database generator. Bypassing the system database generator and/or using the patches to break any and all rules embodied in the database and system configuration that will help you to go down the desired path.
4. ***Use-breakpoints:*** Put breakpoints at the branch points where the hard-to-sensitize path segment begin and then patch the transaction control block to force the path.

5.3.4 Instrumentation

Instrumentation plays an important role in transaction flow testing than in unit testing. Counters may not be useful as the same module might appear in many different flows and the system could be simultaneously processing different transactions. The information of the path taken for a given transaction must be kept with that transaction. It can be recorded either by a central dispatcher or by the individual processing modules. We would need a trace of all the processing steps for the transaction, the queues on which it resided and the entries and exits to and from the dispatcher. Operating system itself provides such traces in some systems. Heavy instrumentation is affordable as compared to the unit testing instrumentation because the overhead of such instrumentation is typically small compared to the processing. Another alternative is to make the instrumentation part of system testing.

5.3.5 Test Databases

30-40% of the efforts of transaction-flow testing go into the design and maintenance of the test databases. Although it is not treated as a major part but can lead to a lot of errors. The first error is to be unaware that there's a test database to be designed. This makes the testers and the developers to design their own databases which are not compatible with each other. Thus, every tester and developer uses the system

exclusively. The resulting tests are configuration sensitive and they cannot be ported from one suite to the other. Thus, to avoid such problems, often testing groups are given this responsibility as the independent testers need more elaborate test setups than programmers.

5.3.6 Execution

The transaction-flow testing mostly requires test execution automation. Even if there are only a few hundred test cases available to achieve C1+C2 transaction-flow coverage, we may be re-running them several times over the entire project duration. Transaction-flow testing with the intention of achieving C1+C2 leads to a big increase in the number of test cases. And these cannot be done rightly without automation.

Check Your Progress 2

State whether the following statements are true or false:

1. Test databases can be created individually by both the tester and the programmer.
2. During the transactional walkthroughs the designers must know what the system is supposed as well as how to implement that functionality.
3. The automation of transactional testing is useful.

5.4 IMPLEMENTATION COMMENTS

5.4.1 Transaction-based Systems

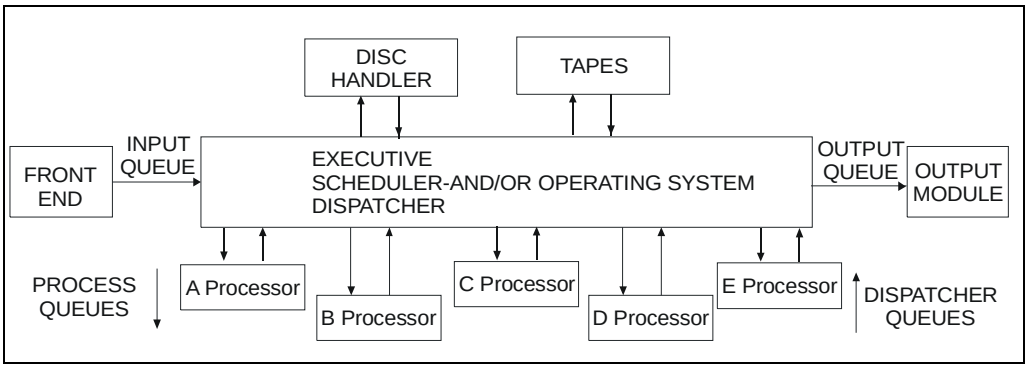


Figure 5.4: Transaction Flow Implementation

Based on the Figure 5.4, in order to make the transaction-flow testing easier, we can use the following things:

1. ***Transaction Control Block:*** A transaction control block is explicitly associated with every live transaction. The block contains the transaction type, identity, processing state and various other things. It might contain this information by itself or might contain pointers to the information. It does not matter how exactly the information is contained, more importantly it is the unique data object through which anything we can know anything about a transaction. The control-flow block is created when the transaction is born and is returned to the pool when the transaction leaves the system for archival storage.
2. ***Central, Common, Processing Queues:*** Transaction control blocks are not passed directly from one process to another, but are transferred from process to process by means of centralized explicit processing queues. The dispatcher links control blocks to processes. Processes link control blocks send the flow back to the dispatcher when they are done.

3. ***Transaction Dispatcher:*** There is a centralized transaction dispatcher. Based on the transaction's current state and transaction type, the next process is determined from stored dispatch tables or a similar mechanism.
4. ***Recovery and Other Logs:*** Key points in the transaction's life are recorded for several different purposes, the most important being, transaction recovery support and transaction accounting.
5. ***Self-test Support:*** The transaction control tables have privileged models that can be used for test and diagnostic purposes. There are special transaction types and states for normal transactions whose only purpose is to facilitate testing.

5.4.2 Hidden Languages

Transactional flow graphs may not be as simple as a control flow-graph. The decisions may be made in a processing module. Or, the dispatcher may direct the flows, contained in the transaction's control block or stored elsewhere in the database, based on control codes. Such codes form an internal language. These languages are often undeclared, undefined, unrecognized and undocumented. Their syntax, semantics and processors cannot be debugged. There can be following shortcomings in these languages:

1. The language is rarely checked for self-consistency.
2. The language may be interpreted either centrally or distributed but may still have bugs.
3. The program may have bugs or may become corrupted by bugs far removed from the transaction under consideration.
4. Any transaction processing module can have bugs.

If transaction control tables are used to direct the flow, it is effective to treat that mechanism as if an actual language has been implemented. We look for the basic components of the language and then document syntax for this primitive, undeclared language and discuss this syntax with whomever responsible for implementing transaction-control structure and software.

5.5 LET US SUM UP

The methods discussed for path testing of units and programs can be applied with suitable interpretation to functional testing based on transaction flows.

Full coverage (C1+C2) is required for all flows, but most bugs will be found on the strange, meaningless, weird paths. Transaction-flow control may be implemented using an undeclared and unrecognized internal language. Get it recognized, get it declared and then test its syntax.

The practice of attempting to design tests based on transaction-flow representation of requirements and discussing those attempts with the designer can unearth more bugs than any tests you run.

5.6 LESSON END ACTIVITIES

1. Discuss the various implementation techniques for transaction based testing.
2. How are hidden languages helpful in transaction testing?
3. Explain the role of walkthroughs and reviews in the transaction based testing.

5.7 KEYWORDS

Transactional flow graph: The graph based on the behavioral model of the program that leads to functional testing.

Transaction: Transaction is a unit of work from a system's point of view. It consists of a sequence of operations part of which is performed by a system, person, or devices out of system.

5.8 QUESTIONS FOR DISCUSSION

1. Explain path sensitizing and instrumentation in the light of transaction-flow testing.
2. What is a transactional flow? What are the various complications it has? How can these be minimized?
3. How is transactional testing different from control-flow based testing?

Check Your Progress: Model Answers

CYP 1

1. (a) False

   (b) True

2. A transaction flow is a representation of a path taken by a transaction through a succession of processing modules.

CYP 2

1. False
2. False
3. True

5.9 SUGGESTED READINGS

Boris Beizer, *Software Testing Techniques*, Second Edition, Dreamtech Press, 2003.

Myers and Glenford, J., *The Art of Software Testing*, John-Wiley & Sons, 1979.

Roger, S. Pressman, *Software Engineering – A Practitioner's Approach*, 5th Edition, McGraw Hill, 2001.

Marnie, L. Hutcheson, *Software Testing Fundamentals*, Wiley-India, 2007.

LESSON

6

DATA FLOW TESTING

CONTENTS

6.0 Aims and Objectives

6.1 Introduction

6.2 Data Flow Testing Basics

- 6.2.1 Data Flow Graphs
- 6.2.2 Data Flow Model

6.3 Data Flow Testing Strategies

- 6.3.1 Terminologies
- 6.3.2 Strategies
- 6.3.3 Slicing, Dicing, Data Flow and Debugging

6.4 Applications, Tools and Effectiveness

6.5 Let us Sum up

6.6 Lesson End Activities

6.7 Keywords

6.8 Questions for Discussion

6.9 Suggested Readings

6.0 AIMS AND OBJECTIVES

After studying this lesson, you would be able to understand:

- The basics of data flow testing – data flow machines, data flow graphs, data flow model
- The strategies of data flow testing – terminologies, strategies, slicing, dicing, debugging
- Applications and its effectiveness

6.1 INTRODUCTION

Data flow testing utilizes the control flow graph to explore the unreasonable things that can happen to data (data flow anomalies). Consideration of data flow anomalies leads to test path selection strategies that fill the gaps between complete path testing and branch and statement testing.

6.2 DATA FLOW TESTING BASICS

Data flow testing is the name given to a family of test strategies based on selecting paths through the programs control flow in order to explore sequences of events related to the status of data objects.

Low-cost computation and memory elements have made possible massively parallel machines that can break the time logjam of current architectures. Most computers today are Von Neumann machines. This architecture allows interchangeable storage of data and instructions in the same memory locations. The Von Neumann architecture executes one statement at a time. These machines have a single execution unit and a single set of registers in which to process. Thus, these machines sequence onto problems that may not necessarily be sequential.

Massively parallel machines (multi-instruction, multi data - MIMD) have multiple mechanisms for executing a problem. They can fetch into several objects and process them in parallel. The decision of parallel computation is left to the compiler. The compiler produces parallel computation instruction for a MIMD machine and sequential instructions for a conventional machine. We can also say that a sequential machine is a special case of a parallel machine with a single processor. Irrespective of this, from the programmers' point of view, it will be data-flow software that will have to be tested.

One of the advantages of a data-flow graph is that there is no restriction to uni-processing as there is for the control flow graph. There are many independent parallel streams which constitute a control flow here.

The bug assumption for data-flow testing strategies is that control flow is generally correct and the something has gone wrong with the software so that data objects are not available when they should be. Also, if there is a control-flow problem, we expect it to have symptoms that can be detected by data-flow analysis.

6.2.1 Data Flow Graphs

The data flow graph is a graph consisting of nodes and directed links i.e. links with arrows on them. We will be doing data flow testing but would not be using data flow graphs as such. We will use an ordinary annotated control flow graph.

Data objects can be created, killed and/or used. They can be used in two different ways in a calculation or as part of a control flow predicate.

d – Defined, created, initialized, etc.

k – killed, undefined, released.

u – used for something.

c – used in a calculation.

p – used in a predicate.

1. ***Defined:*** An object is defined explicitly when it appears in a data declaration. Defined can also mean that a file has been opened, a dynamically allocated object has been allocated, something is pushed onto the stack, a record written, etc.
2. ***Killed or Undefined:*** An object is killed or undefined when it is released or otherwise made unavailable, or when its contents are no longer known with certainty. Define and kill are complementary operations. They come in pairs generally and one does the opposite of the other.
3. ***Usage:*** A variable used for computation (c) when it appears on the right-hand side of an assignment statement, as a pointer, as part of a pointer calculation; a file

record is read or written, and so on. It is used in a predicate (p) when it appears directly in a predicate, but also implicitly as the control variable of a loop, in an expression used to evaluate the control flow of a case statement, as a pointer to an object that will be used to direct control flow.

Data Flow Anomalies

We will discuss only some of the most important data flow anomalies here. An anomaly is the one denoted by a two-character sequence of action. The various possible two-letter combinations possible for d, k and u are as listed under. Some of these are suspicious, some are bugs and some are okay.

dd Probably harmless but suspicious. Why do we need to define an object twice without using it in between?

dk Probably a bug. Why to define an object without using it?

du Normal case. The object is used after defining it.

kd Normal case. An object is killed and then redefined.

kk Harmless but probably buggy. Did you want to be sure it was really killed?

ku A Bug. The object does not exist in the sense that its value is undefined or indeterminate.

ud Usually not a bug because the language permits reassignment at almost any time.

uk Normal situation.

uu Normal situation.

In addition to these, there are six single letter situations. A leading dash means nothing of the interest occurs prior to the action noted along the entry-exit path of interest and a trailing dash to mean that nothing happens after the point of interest to the exit.

-k : possibly anomalous because from the entrance to this point on the path, the variable had not been defined. We're killing a variable that does not exist; but note that the variable might have been created by a called routine or might be global.

-d : okay. This is just the first definition along this path.

-u : Possibly anomalous. Not anomalous if the variable is global and has been previously defined.

k- : Not anomalous. The last thing done on this path was to kill the variable.

d- : Possibly anomalous. The variable was defined and not used on this path; but this could be a global definition or within a routine that defines the variables for other routines.

u- : Not anomalous. The variable was used on this path but not killed. This signals a frequently occurring bug. This means an instance where a dynamically allocated object was not returned to the pool after use.

Data Flow Anomaly State Graph

As per the data flow model an object can be in one of the four distinct states. These are:

K – undefined, previously killed, does not exist.

D – defined but not yet used for anything.

U – Has been used for computation or in predicate.

A – Anomalous.

If a variable has not been defined, it is said to be in the state K. If we use this variable or try to kill it, the state of the variable becomes anomalous (A). Once a variable is anomalous, it cannot be brought back to a normal state by taking any action whatsoever. If it is defined (d), it goes into D, or defined but not yet used state. If it has been defined (D) and redefined (d) or killed without use (k), it becomes anomalous, while usage (u) brings it to the U state. If in U, redefinition (d) brings it to D, u keeps it in U and k kills it.

Static versus Dynamic Anomaly Detection

Static analysis is the one done on source code without executing it e.g. syntax error in code. Dynamic analysis is the one done on the fly as the program is executed and is based on the intermediate values that result from the program's execution e.g. division by zero. If a data flow anomaly can be detected using static analysis method, then it does not belong to testing but to the language processor.

Static analysis is not sufficient and still requires further testing as it is inadequate in the following situations:

1. ***Dead Variable:*** Although we can prove that a variable is dead or alive at a particular point in a program, the general problem is unsolvable.
2. ***Arrays:*** Array is defined or killed as a single object but is referenced using individual locations, which causes problems. Pointers to arrays are generated dynamically, so we cannot do static analysis to validate this pointer.
3. ***Records and Pointers:*** In a lot of problems we create files and their names dynamically and there's no way to determine their whether such objects are in the proper state on a given path or whether they exist at all, without execution.
4. ***Dynamic Subroutine or Function Names in a Call:*** A subroutine or a function name is a dynamic variable in a call. We cannot determine if a call to a subroutine is correct without executing it.
5. ***False Anomalies:*** Anomalies could be path specific. A clear bug may not be a bug if the path along which it exists is unachievable. Such anomalies are called false anomalies. The problem to determine if a path is achievable or not cannot be solved.
6. ***Recoverable Anomalies and Alternate State Graphs:*** Anomaly recovery is possible but only if the language processor has a built-in anomaly definition with which you may or may not agree.
7. ***Concurrency, Interrupts, System Issues:*** Whenever we think in terms of systems and not in terms of single-task uni-processor environment, the anomalies become more complicated. Thus, much of the system testing is aimed at detecting data-flow anomalies that cannot be detected in the context of a single routine.

Although static analysis has limits, they are worth using and prove better for anomaly detection.

6.2.2 Data Flow Model

The data flow model that we are discussing depends on the program's control flow graph. We associate with each link a symbol or a sequence of symbols which denote the sequence of data operations on that link with respect to the variable of interest.

Such annotations are called link weights. The control flow-graph is the same for all variables, only the weights attached to each of the links change.

Components of the Model

Let us discuss some of the data flow graph modeling rules:

1. For every statement there is a node, which is named. Each node has at least one outlink and at least one inlink except for the exit nodes which do not have outlinks and entry nodes, which do not have inlinks.
2. Exit nodes are dummy nodes placed at the outgoing arrow heads of the exit statements to complete the graph. Similarly, entry nodes are dummy nodes placed at entry statements.
3. The outlinks of simple statements are weighted by proper sequence of data-flow actions for that statement.
4. Predicate nodes are weighted with the p-use(s) on every outlink, appropriate to that outlink.
5. Every sequence of simple statements can be replaced by a pair of nodes that has, as weights on the link between them, the concatenation of link weights.
6. If there are several data-flow actions on a given link for a given variable, then the weight of the link is denoted by the sequence of actions on that link for that variable.
7. Conversely, a link with several data-flow actions on it can be replaced by a succession of equivalent links, each of which has at the most one data-flow action for any variable.

Check Your Progress 1

1. State whether the following statements are true or false:
   (a) The sequence ku denotes a bug.
   (b) Data flow graphs cannot represent parallel processing streams.
2. Fill in the blanks:
   (a) Data flow graph is a graph consisting of ..... and ..... links.
   (b) Data objects can be used either in ..... or as part of a .....

6.3 DATA FLOW TESTING STRATEGIES

Data flow testing strategies are structural in nature. Required element testing is the way of generating a family of test strategies based on a structural characterization of the way test cases are to be defined and a functional characterization that test case must satisfy. Pure structural and functional testing as well as hybrid strategies can be employed.

Every algorithm used for selecting links and/or nodes defines a corresponding test strategy. The only structural characteristics used in the path-testing are the raw program-control graphs. Nodes and links are considered to have no property other than the fact that they exist.

In contrast to the path-testing strategies, data-flow strategies take into account what happens to data objects on the links in addition to the raw connectivity of the graph. This strategy requires data-flow link weights and is based on selecting test path

segments (called subpaths) that satisfy some characteristic of data flows for all data objects. Based on the rule of selecting test path segments, a major objective of testing research is to determine the relative strength of the strategy corresponding to that rule i.e., to find out whether it is stronger or weaker than some strategy or incomparable. A strategy X is stronger than strategy Y if all test cases produced under Y are included in those produced under X – conversely weaker.

6.3.1 Terminologies

Assuming that all paths are achievable let us define some terminologies:

1. A definition-clear path segment with respect to variable X is a connected sequence of links such that X is defined on the first link and not defined or killed on any subsequent link of that path segment. A definition-clear path between two nodes does not imply that all subpaths between those nodes are definition-clear. There are many subpaths between nodes some which could have definition on them and some not. A definition clear path segment does not prevent loops.
2. A loop-free path segment is a path segment for which every node is visited at most once.
3. A simple path is a path segment in which at most one node is visited twice. A simple path segment is either loop-free or if there is a loop, only one node is involved.
4. A du path from node i to k is a path segment such that if the last link has a computational use of X, then the path is simple and definition-clear; if the penultimate node is j, i.e. the path is (i,p,q,r,s,…..,j,k) and link (j,k) has a predicate use –then the path from I to j is both loop-free and definition-clear.

6.3.2 Strategies

The strategies that we will be discussing here are based on the program's control flow-graphs. The extent of usage of the predicate and computational uses in the test set differentiates them. They may also differ depending upon if all the paths of a given type are required or only one path of that type, etc.

1. ***All-du paths (ADUP):*** It is the strongest of all data-flow testing strategies. It requires that every du path from every definition of every variable to every use of that definition be exercised under some test. This strategy requires the greatest number of paths for testing.
2. ***All-Uses Strategy:*** At least one path from every definition of every variable to every use of that can be reached by that definition. For every use of the variable, there is a path from the definition of that variable to the use.
3. ***All-p-Uses/Some-c-Uses and All-c-Uses/Some-p-Uses Strategies:***
   - *APU+C Strategy:* For every variable and every definition of that variable, include at least one path from the definition to every predicate use; if there are definitions of that variable that are not covered then add computational use test cases as required to cover every definition. In this testing strategy, there is a path from every definition to every p-use of that definition. If there is a definition with no p-use following it, then a c-use of the definition is covered.
   - *ACU+P Strategy:* For every variable and every definition of that variable, include at least one path from the definition to every computational use; if there are definitions of the variable that are not covered then add predicate use test cases as required to cover ever definition. In this testing strategy, for

every variable, there is a path from every definition to every c-use of that definition. If there is a definition with no c-use following it, then a p-use of the definition is considered.

4. ***All Definitions Strategy:*** Every definition of every variable be covered by at least one use of that variable, be that use a computational use or a predicate use. In this strategy, there is path from every definition to at least one use of that definition.

5. ***All-Predicate-Uses, All-Computational Uses Strategies:*** All p-uses strategy (APU) is derived from APU+C by dropping the requirement of including a c-use if there are no p-use instances following the definition. In this testing strategy, for every variable, there is path from every definition to every p-use of that definition. If there is a definition with no p-use following it, then it is dropped from the contention.

ACU strategy is derived from ACU+P by dropping the requirement of including a p-use if there are no c-use instances following the definition. In this testing strategy, for every variable, there is a path from every definition to every c-use of that definition. If there is a definition with no c-use following it, then it is dropped from contention.

Let us discuss all these strategies with the help of an example. In this example, we calculate the monthly bill of a mobile phone user. The following code snapshot talks about the actual billing logic:

```
bill = 0.0;
if (usage > 0)
   bill = 40;
if (usage > 100)
{
   if (usage < = 200)
   {
      bill = bill + (usage – 100) * 0.5;
   }
   else
   {
      bill = bill + 50 + (usage – 200) * 0.1;
      if (bill >= 100)
      {
         bill = bill * 0.9;
      }
   }
}
return bill;
```

The control flow diagram for the program given above is as shown in Figure 6.1.

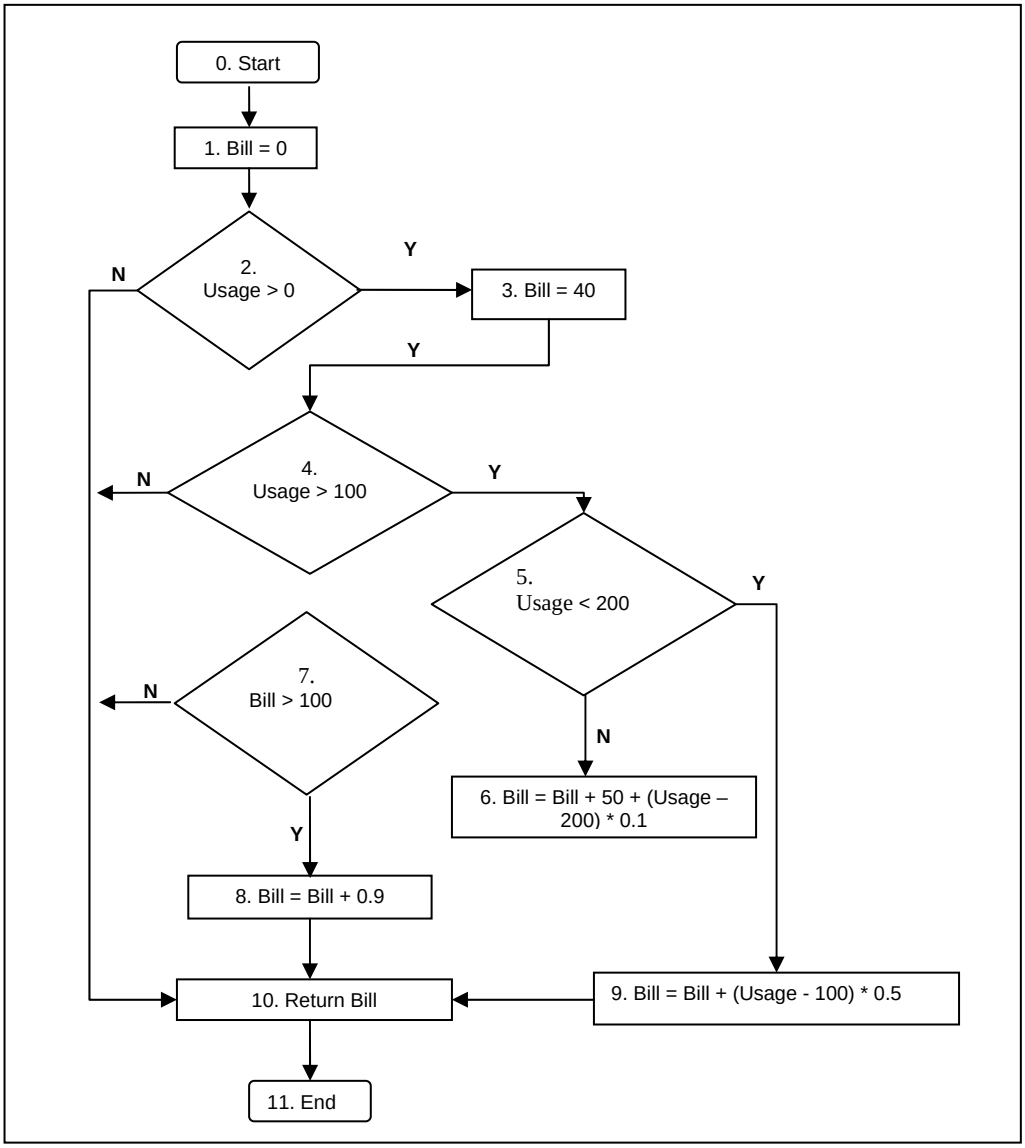


Figure 6.1: Control Flow Diagram

The annotated control flow graph for variable ‘bill’ is as shown in Figure 6.2 below.

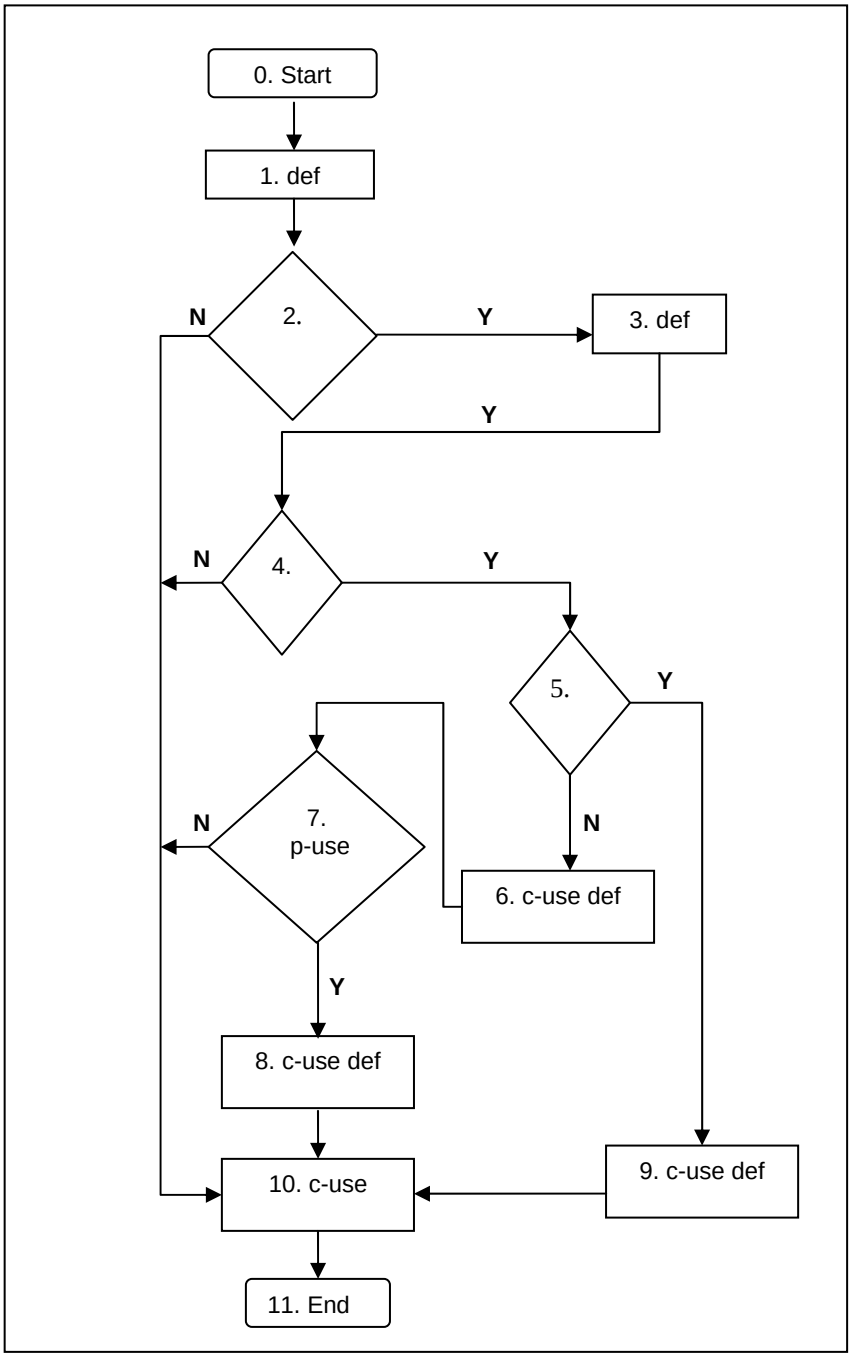


Figure 6.2: Annotated Control Flow Diagram for Variable ‘Bill’

The annotated control flow graph for variable ‘bill’ is as shown in Figure 6.3 below.

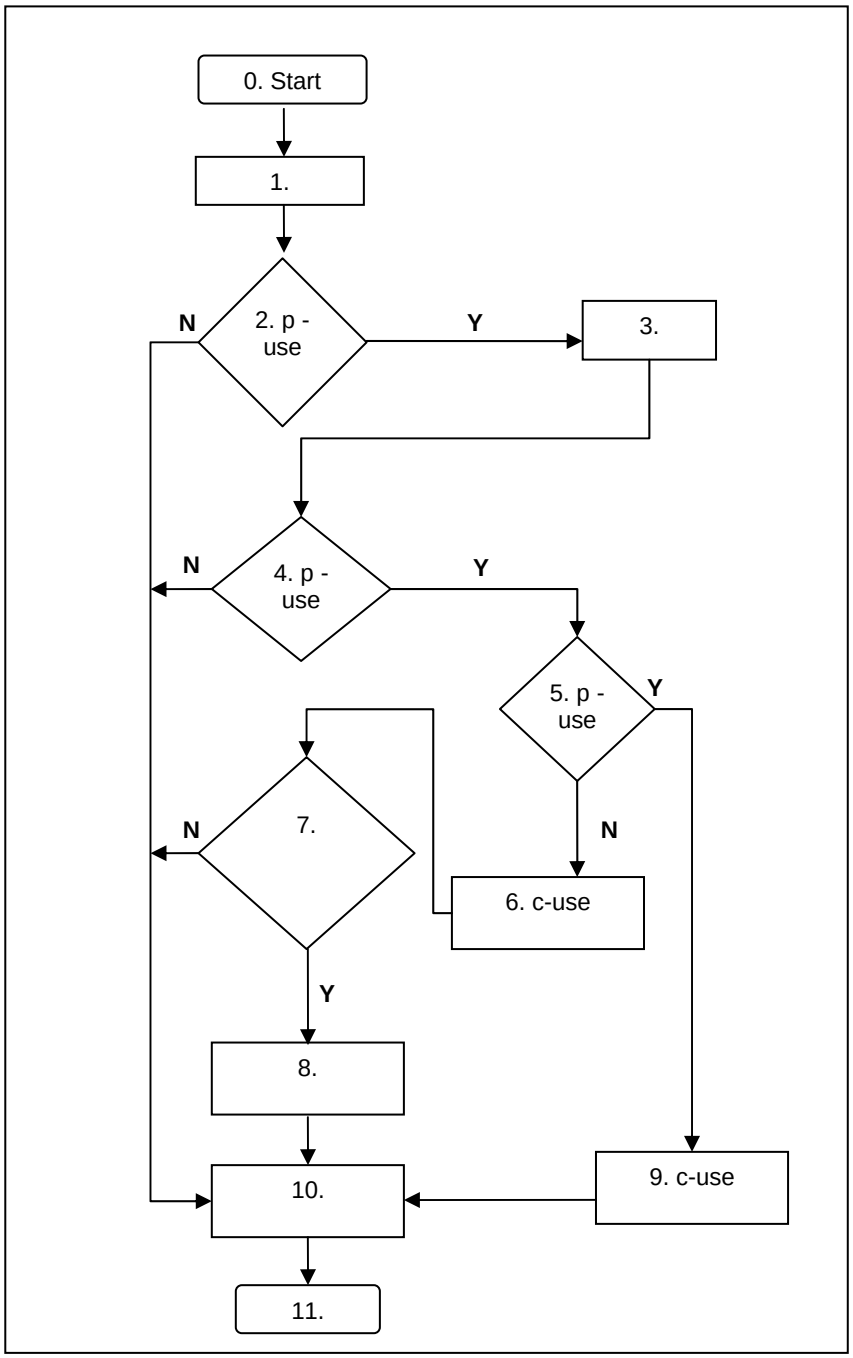


Figure 6.3: Annotated Control Flow Graph for Variable ‘Usage’

Ordering of Strategies

For selection of test cases, we need to analyze the relative strength of the strategies of data-flow testing. Figure 6.4 depicts the relative strength of the data-flow strategies and other control-flow testing strategies such a branch and all-statement. According to this figure, the strength of testing strategies reduces along the direction of the arrows. Thus, ALL PATHS is the strongest testing strategy. Also, ACU+P and APU+C run parallel and hence are comparable.

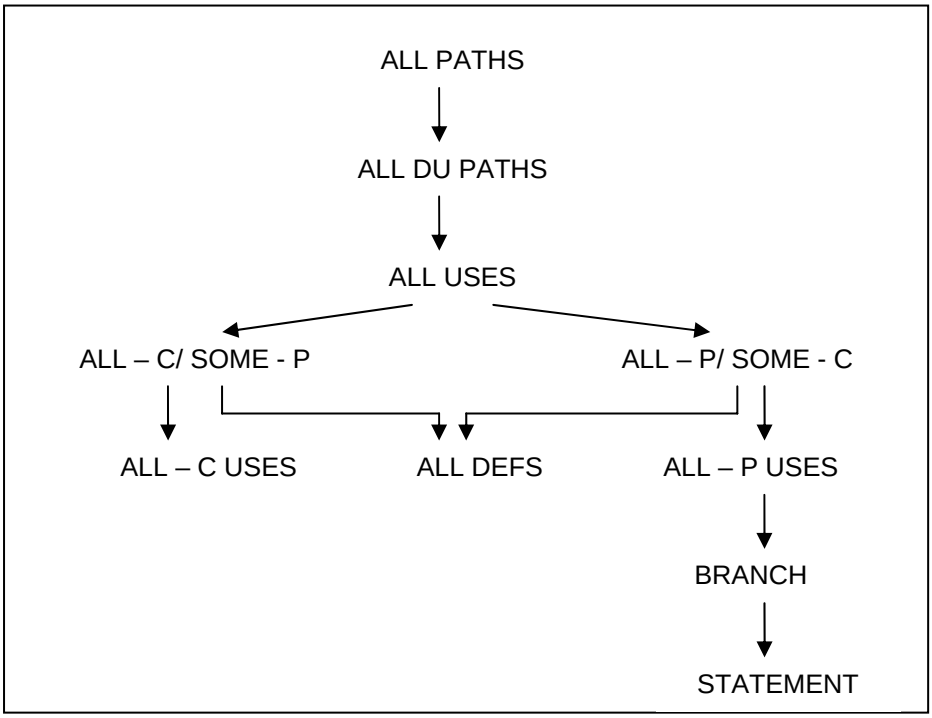


Figure 6.4: Relative Strength of Testing Strategies

6.3.3 Slicing, Dicing, Data Flow and Debugging

A program slices is a part of the program defined with respect to a given variable X and a statement i: it is the set of all statements that could affect the value of X at statement i – where the influence of a faulty statement could result from an improper computational use or predicate use of some other variables at prior statements. If X is incorrect at statement i, it follows that the bug must be in the program slice for X with respect to i. A program dice is a part of slice in which all statements which are known to be correct have been removed. Alternatively, a dice is obtained from a slice by incorporating information obtained through testing or experimenting. These procedures of slicing and debugging are at the heart of a debugger. The debugger limits his scope to the statements that have caused the faulty value at statement i and then eliminates the statements that have proven to be incorrect while testing. Debugging is an iterative process and ends when the dice has been reduced to the one faulty statement.

Dynamic slicing is a refinement of static slicing in which only statements on achievable paths to the statement in question are included.

Check Your Progress 2

1. Fill in the blanks:
   (a) ..... is the strongest data flow testing strategy.
   (b) There is path from every definition to at least one use of that definition in the ..... strategy.
2. State whether the following statements are true or false:
   (a) A simple path is a path segment in which at most one node is visited twice.
   (b) Data integrity is an important code as integrity.

6.4 APPLICATIONS, TOOLS AND EFFECTIVENESS

Sneed's comparison of branch coverage to data-flow testing is that the number of bugs detected by requiring 90% data coverage is twice as high as those detected by requiring 90% branch coverage.

The study conducted by Weyuker on the comparison of data flow testing strategies is the most thorough to date. The study examined the number of tests needed to satisfy ACU, APU, AU and ADUP. The number of test cases is normalized to the number of binary decisions in the program.

Data flow testing concepts have been around for a long time. It is as cost effective as the branch coverage and statement. Finding of data-flow-covering test sets has more or less the same difficulty level as finding branch-covering test sets.

Data flow testing entails additional record keeping, for which a computer is most effective. Simpler tools, that can keep track of which variables were defined and where and in which subsequent statement the definition is used, can significantly reduce the efforts of data-flow testing. Test designs are similar to that of P1 and P2 as the one entry/exit test case passes through a bunch of definitions and uses different variables.

6.5 LET US SUM UP

Data integrity is as important as code integrity. All data definitions and subsequent uses must be tested. What forms a data flow anomaly is peculiar to the application. Use the available tools to detect those anomalies that can be detected statistically. Static data flow anomaly detection can be an important criterion in selecting a language processor.

The data flow testing strategies reduce the gap between all paths and branch testing. Out of all the strategies available, AU has the best value for money, although it requires double the test cases as for branch testing. AU can be achieved using tools available. The symbols d, k, u and associated anomalies can be interpreted in terms of file opening and closing, resource management and other applications.

6.6 LESSON END ACTIVITIES

1. Differentiate between static and dynamic anomaly detection.
2. Mention the various states of a data object and their usage.
3. List down the various data flow anomalies. What are the different alphabets used to represent the same?

6.7 KEYWORDS

Branch Coverage: Branch coverage is achieved when every path from a control flow graph node has been executed at least once by a test suite.

Data-flow Anomaly: A data-flow anomaly is denoted by a two-character sequence of actions.

Data-flow Testing: It selects paths through the program's control flow in order to explore sequences of events related to the status of data objects.

State: The state of an object can be defined as a set of instance variable value combinations that share some property of interest.

Statement Coverage: Coverage achieved when all the statements in a method have been executed at least once.

6.8 QUESTIONS FOR DISCUSSION

1. List the different strategies of data-flow testing.
2. Compare the effectiveness of the various strategies for data-flow testing.
3. Describe a model used for data flow testing.

Check Your Progress: Model Answers

CYP 1

1. (a) True
   (b) False
2. (a) calculation, control flow predicate
   (b) nodes, directed

CYP 2

1. (a) All paths
   (b) all Definitions
2. (a) True
   (b) True

6.9 SUGGESTED READINGS

Boris Beizer, *Software Testing Techniques*, Second Edition, Dreamtech Press, 2003.

Myers and Glenford, J., *The Art of Software Testing*, John-Wiley & Sons, 1979.

Roger, S. Pressman, *Software Engineering – A Practitioner's Approach*, 5th Edition, McGraw Hill, 2001.

Marnie, L. Hutcheson, *Software Testing Fundamentals*, Wiley-India, 2007.

LESSON

7

SYNTAX TESTING

CONTENTS

7.0 Aims and Objectives

7.1 Introduction

7.2 Why, What and How?

- 7.2.1 Garbage
- 7.2.2 Casual and Malicious Users
- 7.2.3 Operators
- 7.2.4 The Internal World
- 7.2.5 What to Do?
- 7.2.6 Applications and Hidden Languages
- 7.2.7 The Graph we Cover
- 7.2.8 Overview

7.3 A Grammar for Formats

- 7.3.1 Objectives
- 7.3.2 BNF Notation (BACK59)

7.4 Implementation and Applications

- 7.4.1 Execution Automation
- 7.4.2 Design Automation
- 7.4.3 Productivity, Training and Effectiveness
- 7.4.4 Ad-Lib Tests

7.5 Testability Tips

- 7.5.1 The Tip
- 7.5.2 Compiler Overview
- 7.5.3 Typical Software
- 7.5.4 Separation of Phases
- 7.5.5 Prerequisites

7.6 Let us Sum up

7.7 Lesson End Activities

7.8 Keywords

7.9 Questions for Discussion

7.10 Suggested Readings

7.0 AIMS AND OBJECTIVES

After studying this lesson, you would be able to understand:

- The why, what and how of syntax testing – garbage, operators, malicious and casual users, applications and hidden languages, etc.
- A grammar for formats – Its objectives and the BNF notation
- Implementation and Application – Execution automation, design automation, productivity, training and effectiveness and Ad-Lib tests
- Testability tips – The tip, compiler overview, typical software, separation of phases, prerequisites

7.1 INTRODUCTION

System inputs must be validated. Internal and external inputs must comply with the format Backus-Naur Form – a specification form that can be mechanically converted into more input-data validation tests that anyone could want to execute.

7.2 WHY, WHAT AND HOW?

7.2.1 Garbage

Garbage-in equals Garbage-out is a situation when the program screws up hurting the sentiments of the people involved. It leads to an investigation to know the cause of this occurrence. The reason for this is the failure to install good data-validation checks, or worse, or failure to test the system's tolerance for bad data. Garbage should not get into the system, in the first place.

7.2.2 Casual and Malicious Users

Systems that have to be used by public must be robust and must validate the input. The more the users using the system, the likelier it is that they would hit the point which is vulnerable to bad inputs.

Malicious users are those who delight in doing strange things to our systems. They could as well be programmers. They are persistent and systematic. One attack by them is severe than years of usage by ordinary users and bugs found by chance.

7.2.3 Operators

There are operators of the system which use the system in a way in which it was not intended to resulting in eventualities. They commit mistakes which can be serious. They work using intuition, common sense, and brilliance to find a chance to prove the system malfunctioning.

7.2.4 The Internal World

Not only can the big system be attacked by external environment but also a hostile internal environment. A huge system can be divided into numerous loosely coupled subsystems with various internal interfaces. These interfaces present the opportunity for data corruption and might require explicit data validations. Failing hardware can put bad data into memory, across channels, etc.

7.2.5 What to Do?

We can at first validate the input to defend the system against the hostile world. A good designed system will not accept the garbage at all. As part of system testing,

we perform input-tolerance testing as if it is being done in the final phase, so it is done by independent testers.

7.2.6 Applications and Hidden Languages

Syntax testing can be applied in most of the systems as they have hidden languages. A hidden language is a programming language that has not been recognized as such. Syntax testing is used to validate and break the explicit or implicit parser of this language. The problem with these hidden languages is that they don't have a formal definition of syntax, syntax has bugs and parsing is often intertwined with processing. These languages can be exploited by recognizing them using the following techniques:

1. User and operator commands are examples of languages.
2. The counterparts, to the operator and user command language for batch processing are job control languages at the operating system level or at the application-specific level.
3. An offline database generator is used to create the database.
4. A system wide inter-process communication convention has been established.
5. The format used for inter-process communication does not consist of simple and fixed fields.

Syntax testing is a method which depends on creating a lot of test cases. It is an effective method because the number of test cases designed is large which makes it easier and more likely to reveal bugs.

7.2.7 The Graph we Cover

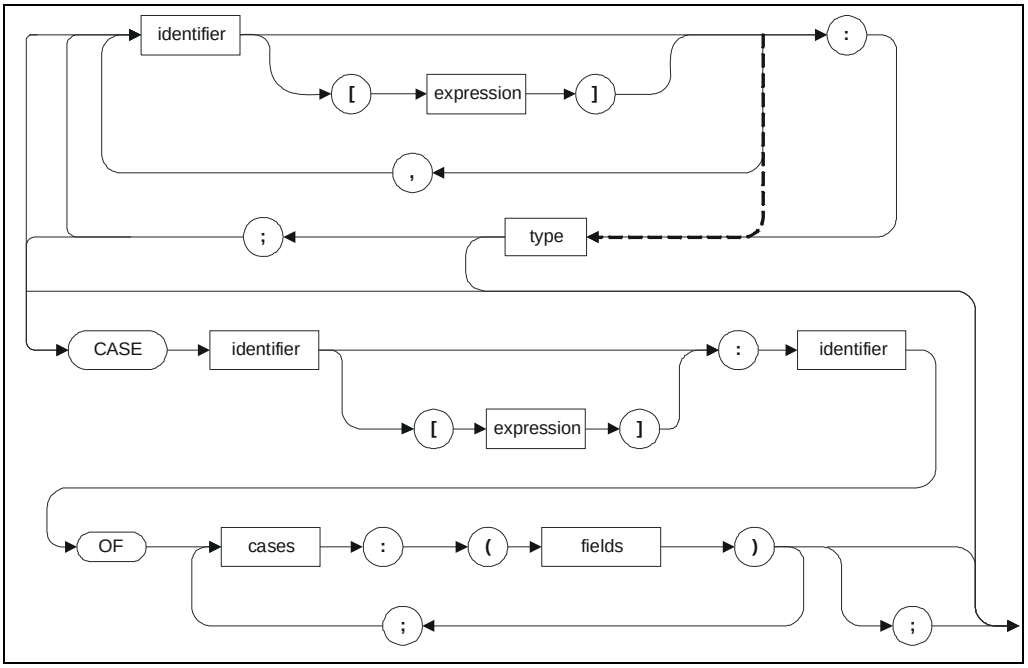


Figure 7.1: Pascal Fields Definition Graph (Courtesy Microsoft)

We can perform the syntax testing using the graph shown in Figure 7.1. We can define a set of covering paths through this graph. For each path, generate the fields corresponding to that path. We also cover all the loops in the graph during syntax testing.

7.2.8 Overview

Syntax testing consists of the following steps:

1. Identification of the target language or format.
2. Define the syntax of the language, formally, in a convenient notation such as Backus-Naur Form (BNF).
3. Test and debug the syntax to assure that it is complete and consistent and that it satisfies the intended semantics.
4. While covering the syntax graph we must be sure that we have tested all the options.
5. Syntax testing must be automated to the best possible extent.

Check Your Progress 1

1. Define Backus Naur Form (BNF).

   .....

   .....

2. State whether the following statements are true or false:

   (a) Syntax testing can be applied in systems having hidden languages.

   (b) At times it is required to inject garbage into the system to test its credibility.

7.3 A GRAMMAR FOR FORMATS

7.3.1 Objectives

Every input has syntax. Data validation checks the input for correct syntax. The syntax is best defined in a formal language which can be used by the tester to create useful garbage. This specification can be conveniently expressed in Backus Aur Form which is very similar to the regular expressions.

7.3.2 BNF Notation (BACK59)

Elements

Every input can be considered as a string of characters. The software accepts valid strings and rejects the invalid ones. If the software fails on a string then we have found the bug, but if it accepts then it's guilty of GIGO. Let us consider a sample definition.

Alpha_characters:

A/B/C/D/E/F/G/H/I/J/K/L/M/N/O/P/Q/R/S/T/U/V/W/X/Y/Z

Numerals ::= 1/2/3/4/5/6/7/8/9

Zero ::= 0

Signs ::= !/@/#/$/%/^/&/*/(/)/-/+/=/'/"/./?/;/:

sSpace ::= sp

At the left hand side is the name of the object to which is assigned the values of the expressions on its right. The symbol '::=' is considered as a single symbol which serves as "is defined as". The slash '/' means "or". We are using the BNF to define a miniature language called the metalanguage. We use sp to refer to the blank spaces.

An italicized symbol is used for a character that cannot be printed conveniently, e.g. null (nl), end-of-text (eot), clear-screen, carriage return (cr), line feed (lf), etc.

BNF Operators

The operators that are used are: +, concatenate, *, +, A^n (exponential), etc. Each definition in turn may refer to another definition or to itself in such a way that eventually it gets down to the characters that form the input string. For example:

Word ::= alpha_character alpha_character / numeral sp numeral

As per this definition, we have the following examples of words and nonWords.

Words: AB, GH, IK, 4 sp 9, 1 sp 5, 8 sp 3

nonWords: GGG, A sp J3, 111, STEP, +, NOT sp DART

The designer wants to detect and accept the words and reject nonWords; the tester wants to generate nonWords and force the program to deal with them.

Repetitions

$Object^{1-3}$ means one to three objects, object* means zero or more occurrences of the object without limit and $object^+$ means one or more repetition of the object. Because both + and * refer to unlimited repetitions, they cannot appear in any of the valid syntax. They cannot be done in a finite memory or in the real world. The software must have means to limit the number of repetitions. This can be done by associating an explicit test with every + or * operator, in which case you should replace the operator with a number. Another way to limit the repetition is by placing a global limit on the length of any string. Yet another way is to limit a common resource such as stack or array size.

The sign of weak software is the ease with which you can destroy it by overloading its repetition-limiting mechanisms. If the mechanism doesn't exist, you can probably scribble all over the stack or code.

Example of a Telephone Number

Special_digit ::= 1/2/5

Zero ::= 0

Other_digit ::= 3/4/6/7/8/9

Ordinary_digit ::= special_digit / zero / other_digit

Exchange_part ::= $other\_digit^2$ ordinary_digit

Number_part ::= $ordinary\_digit^4$

Phone_number ::= exchange_part number_part

According to this definition, following are the examples of phone_numbers:

3469900, 3300000, 9904567

And these are not phone_numbers:

1212555, 55510000, GHJLKF000067%, 0009-456A

In order to reduce the number of steps in the definition, it is useful to enclose an expression in parentheses.

Example of an Operator Command

Operator command = mnemonic field $unit^{1-8}$ +

An operator command consists of a *mnemonic* followed by one or eight field_units and a plus sign.

Field-unit	::= filed delimiter
Mnemonic	::= first_part second_part
Delimiter	::= sp / , / . / $ / *sp$^{1-42}$
Field	::= numeral/alpha/mixed/control
First_part	::= a_vowel a_consonant
Second_part	::= b_consonant
A_vowel	::= A/E/I/O/U
A_consonant	::= B/D/F/G/H/J/K/L/M/N/P/Q/R /S/T/V/W/X/Y/Z
B_consonant	::= B/G/X/Y/Z/W/M/R/C
Alpha	::=a_vowel/a_consonant/ b_consonant
Numeral	::= 1/2/3/4/5/6/7/8/9/0
Control	::= $/*/%/sp/@
Mixed	:= control alpha control/control numeral control/control control control

Following are some valid operator commands:

ABWX A. B. C. 7. +

UAMT W sp sp sp sp +

Following are not operator_commands

ABC sp +

A sp BCDEFGHIJKLMNOPQR sp^{47+}

The telephone number example started with recognizable symbols and constructed some more complicated components from them using the bottom-up approach. The command example started at the top and worked down to the real characters, following a top-down approach.

7.4 IMPLEMENTATION AND APPLICATIONS

7.4.1 Execution Automation

Syntax testing must necessarily have automated test execution because it is not easy to design so many tests manually and it is made easier if done in an automated manner. Syntax testing is effective only if it has a lot of test cases defined.

An automation platform is a prerequisite to execute automation. This can be made out of a PC with hard-disk and general purpose emulator software such as CROSSTALK MK-4.

Manual Execution

A manual execution of test cases is a strict no-no. It is even worse than putting test cases on paper tape and then running the tape in turn.

Capture/Relay

A capture/relay is a system which captures your keystrokes and stuff sent to the screen and stores them for later execution. You execute your designed syntax test cases the first time through these capture/replay systems. These systems have a built-in editor

and pass the test data to a word processor for editing. In this way, even if your first execution is faulty, you will be able to correct it.

Drivers

A driver is a program that automatically sequences through a set of test cases usually stored as data. It can either be built on your own or bought from the market.

Scripting Language

A scripting language is a language used to write test scripts. CASL is a nice scripting language because it can be used to emulate any interface, work from strings stored as data, and provide smart comparisons for test outcome validation, editing and capture/replay.

7.4.2 Design Automation

Syntax testing is a nice method to begin with test automation effort because it's so easy and has high paying results in the first go.

Primitive Methods

Primitive methods involve the use of a copy machine and a typist and later on a word processor. If you understand these primitive methods, then you'll understand how to automate much of syntax test design.

Scripting Language

A scripting language and processor such as CASL has the necessary features required to automate the replacement of good substrings by bad ones on the fly. It is a good technique if the main purpose is to stress the software rather than to validate the format validation software. If we want to do it right, irrespective of language, you need to be more sophisticated.

Random String Generator

We cannot use only random string generators because random strings get recognized as invalid very easily and even a weak front end will catch most bad strings. The probability to hit the vulnerable point is too low. A random string generator is too easy to build. We need to be careful about the place to put the string terminators like carriage returns.

7.4.3 Productivity, Training and Effectiveness

It is pretty easy for even new persons to develop syntax test design and test cases. The people with no experience of test designing before can produce a huge number of syntax test cases. It is also easy to convince a novice tester that testing is infinite with the help of syntax testing.

7.4.4 Ad-Lib Tests

Ad-lib tests are those which test the software for invalid input syntax. These types of tests enhance the confidence in the system as well as the tests. Let us now see what exactly happens in these tests:

1. Most of the ad-lib tests would be input strings violating the format so that the system rejects them.
2. The rest are the good strings that appear to be bad. The system accepts the strings and does as it was told to but is not recognized by the ad-lib tester.

3. A few correct appearing strings will be correctly rejected because of a correlation problem between two field values or a state dependency.
4. Once an ad-lib test proves that the system is wrong, it would be required to dig all the documentation available to ensure if the system behaves as per customer's expectations.

This type of testing is not useful if the system is good, has been tested thoroughly during unit testing and there has been good quality control.

Check Your Progress 2

Fill in the blanks:

1. ..... Tests test the software for invalid input syntax.
2. In BNF, '/' is used for the ..... operation.

7.5 TESTABILITY TIPS

7.5.1 The Tip

Here are a few tips for testing:

1. Bring the hidden language out of the closet.
2. Define the syntax in BNF.
3. Simplify the syntax definition graph.
4. Build a parser.

7.5.2 Compiler Overview

Compiler works in three steps: lexical analysis, parsing and code production. Although we don't use a compiler as often as an interpreter but because we are testing hidden languages, we would be interested in a compiler. Let us now discuss these steps in detail now:

1. ***Lexical analysis:*** This phase does the following:
   a. The analyzer identifies individual fields.
   b. The analyzer identifies the inter-field separators or delimiters.
   c. Classify the field as integer, string, operator, keyword, etc. Some fields are translated at this point like numbers or strings.
   d. New variable names and program labels are stored in a symbol table and replaced by a pointer to that table. This pointer is an example of a token.
   e. Keywords are also replaced by tokens. Numbers and strings are also put in a table and replaced by pointers.
   f. Delimiters are eliminated wherever possible. If the language allows multiple statements per line and there are statement delimiters, statements will be separated so that subsequent processing will be done one statement at a time. Similarly, multiple line statements are combined into a single string.
   g. The output of this phase is the partially translated version of the source in which all linguistic components are replaced by tokens. This act of replacing components by tokens is called tokenizing.

2. ***Parsing:*** Parsing is done on the tokenized strings. The strategy to be used for parsing depends upon the kind of statement, language and the compiler's objective. The validation in the parsing step shows that the string to be parsed corresponds to a path in the syntax graph. The output of the parser is a tree with the statement identifier at the top, primitive elements at the bottom and with intermediate nodes corresponding to definitions that were traversed along the path through the syntax graph.
3. ***Code production:*** It consists of scanning the above tree in such a way to assure that all objects are available when they are needed and then replacing the tokens with sequences of instructions that accomplish what the tokens signify.

7.5.3 Typical Software

Typical software follows the following steps for syntax validation and command processing:

lex_a_little + parse_a_little + process_a_little

Because all these aspects are intertwined with one another, a single bug can involve all these three aspects.

7.5.4 Separation of Phases

Lexical parsing separation means that test strings with combined lexical and syntax errors will not be productive. Parsing-processing separation means that we can separate domain testing from syntax testing. Domain analysis is the first stage of processing and follows parsing, and it is therefore independent of syntax. The bottom line of separation process is the elimination of double-errors and high-order vulnerabilities and therefore the need to even consider such cases.

Separation means separate maintenance. That is, if a processing routine is wrong, there's no need to change the lexical analyzer or parser for that. If a new command is to be added, chances are that only the parser and keyword table will be affected.

All this can come at the cost of more memory and more processing time or may be neither of the two.

7.5.5 Prerequisites

The language should be decent enough so that is possible to do lexical analysis before parsing and parsing before processing. This means that it is possible to pull out the token in a left-to-right manner over the string and to do it independently of any other string or statement in the language. This is an example of a context-dependent language. Languages with more virtuous properties are called context-free.

7.6 LET US SUM UP

Syntax testing commences with a validated format specification. Express the syntax in a formal language like BNF. Simplify the syntax definition graph before designing. Design syntax tests level by level from top to bottom making only one error at a time, one level at a time, leaving everything else as correct as possible.

Test the valid cases covering the definition graph. Concentrate on delimiters and their errors that could cause syntactic ambiguities. Stress all BNF exponent values as for loop testing. Test field-value errors and state dependencies by domain testing and state testing, as appropriate.

Take the design advantage to simplify tests and vice versa. Document copiously and automate as much as possible – use capture/replay systems and editors to create tests

and build or buy drivers to run them. Give attention to the ad-lib testers and remember that they can be replaced by a random string generator. Cover all the valid cases.

7.7 LESSON END ACTIVITIES

1. Explain the Backus Naur Form. Give the complete description of its characters, operators, etc.
2. What is a compiler? How does it work?
3. List down the different techniques of design automation.

7.8 KEYWORDS

Tokens: A token is strictly a string of characters that the compiler replaces with another string of characters.

Tokenizing: The act of replacing components by tokens is called tokenizing

Hidden Language: A hidden language is a programming language that has not been recognized as such.

GIGO: Garbage in Garbage Out

7.9 QUESTIONS FOR DISCUSSION

1. Define capture/reply, drivers and scripting languages.
2. What are the steps to identify a hidden language?
3. List down the various steps included in syntax testing.

Check Your Progress: Model Answers

CYP 1

1. Backus Naur Form is a notation to represent the syntax of a language. It can be used by testers to create useful garbage.
2. (a) True
   (b) True

CYP 2

1. Ad-lib
2. OR

7.10 SUGGESTED READINGS

Boris Beizer, *Software Testing Techniques*, Second Edition, Dreamtech Press, 2003.

Myers and Glenford, J., *The Art of Software Testing*, John-Wiley & Sons, 1979.

Roger, S. Pressman, *Software Engineering – A Practitioner's Approach*, 5th Edition, McGraw Hill, 2001.

Marnie, L. Hutcheson, *Software Testing Fundamentals*, Wiley-India, 2007.

UNIT IV

LESSON

8

LOGIC BASED TESTING

CONTENTS

8.0 Aims and Objectives

8.1 Introduction

8.2 Motivational Overview

- 8.2.1 Hardware Logic Testing
- 8.2.2 Specification Systems and Languages
- 8.2.3 Knowledge based Systems
- 8.2.4 Overview

8.3 Decision Tables

- 8.3.1 Definitions and Notation
- 8.3.2 Decision Table Processors
- 8.3.3 Decision Tables as a Basis for Test Case Design
- 8.3.4 Expansion of Immaterial Classes
- 8.3.5 Test Case Design
- 8.3.6 Decision Tables and Structure

8.4 Path Expressions

- 8.4.1 Boolean Algebra
- 8.4.2 Boolean Equations

8.5 KV Charts

- 8.5.1 The Problem
- 8.5.2 Simple Forms
- 8.5.3 Three Variables
- 8.5.4 Four Variables and More

8.6 Specifications

- 8.6.1 Finding and Translating Logic
- 8.6.2 Ambiguities and Contradictions
- 8.6.3 Don’t-Care and Impossible Terms

8.7 Let us Sum up

8.8 Lesson End Activities

8.9 Keywords

8.10 Questions for Discussion

8.11 Suggested Readings

8.0 AIMS AND OBJECTIVES

After studying this lesson, you would be able to understand:

- Motivational overview – Programmers and Logic, Hardware Logic Testing, Specification Systems and Languages, Knowledge based systems, overview
- Decision tables – definitions and notations, decision table processors, decision tables as basis for test case design, test case design, decision tables and structure
- Path Expression – Boolean algebra and equations
- KV charts – simple forms, three variable, four and more variables
- Specifications – finding and translating logic, ambiguities and contradictions, don't care and impossible terms

8.1 INTRODUCTION

The functional requirements of many programs can be specified using decision tables which provide a useful basis for program and test design. Consistency and completeness can be analyzed using Boolean algebra which can also be used for test design. Boolean algebra is trivialized by using Karnaugh - Veitch charts.

8.2 MOTIVATIONAL OVERVIEW

"Logic" is a commonly used word with programmers. We will learn about logic in its simplest form, i.e. Boolean algebra, and its applications to program and specification test and design. Boolean algebra is to logic as arithmetic is to mathematics. In its absence the tester or programmer is cut off from many test and design and tools that include those techniques.

8.2.1 Hardware Logic Testing

Hardware logic design and hardware logic test design are now intensely automated. Many test methods developed for hardware logic testing can be applied for software logic testing.

8.2.2 Specification Systems and Languages

With the improvements in programming and testing techniques, the bugs have moved closer to the process front end requirements and specifications. These bugs form approximately 8% to 30% of the total and are the costliest as they are the first ones to enter and last ones to leave.

The specifications are hard to express. Boolean algebra or sentential calculus is the most basic of all logic systems. Higher order logic systems are needed and used for formal specifications. But even these advanced methods are based on the Boolean algebra which reasons the importance to understand it.

8.2.3 Knowledge based Systems

The knowledge based system or expert system or artificial intelligence system are preferred today for problems that were once considered to be difficult. These systems include knowledge from a particular domain like medicine, law, civil engineering, etc. This data is then queries and interacted with to build solutions to the problems. We include the knowledge of an expert into a set of rules. The user can then provide data and ask questions based on this data. The user data is processed to obtain conclusions, using a program called inference engine. While testing the knowledge

based systems it is essential to test the validity of the expert's knowledge and the correctness of transcription of that knowledge into a rule base.

8.2.4 Overview

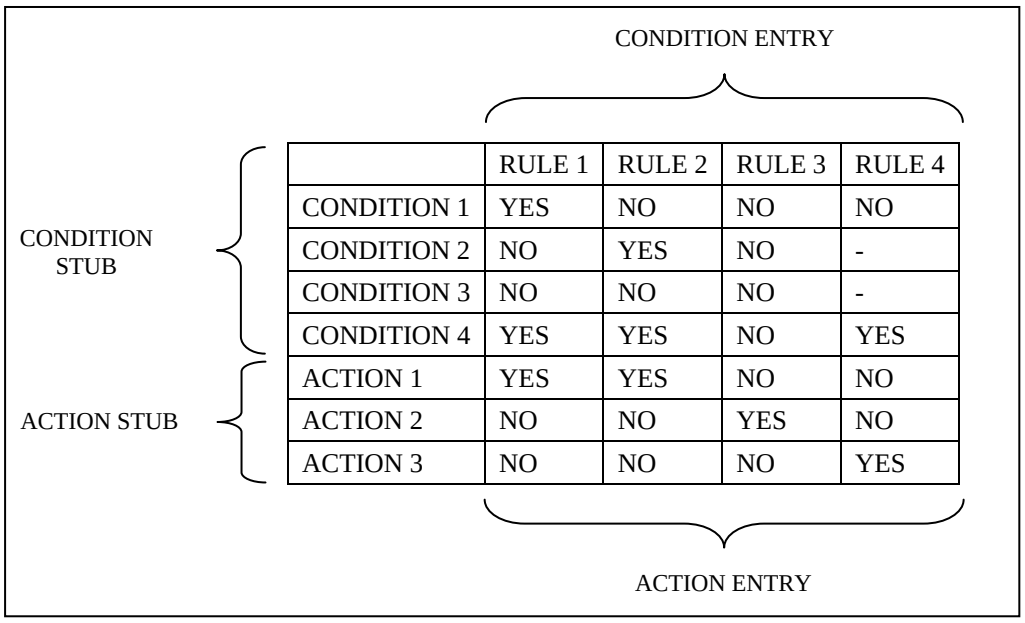
We would begin with decision tables as they are widely used in business data processing and Boolean algebra is embedded in the implementation of these processors.

We can reap maximum benefits of the Boolean algebra manually using the right conceptual tools, the Karnaugh-Veitch diagram.

8.3 DECISION TABLES

8.3.1 Definitions and Notation

Table 8.1: Decision Table



	RULE 1	RULE 2	RULE 3	RULE 4
CONDITION 1	YES	NO	NO	NO
CONDITION 2	NO	YES	NO	-
CONDITION 3	NO	NO	NO	-
CONDITION 4	YES	YES	NO	YES
ACTION 1	YES	YES	NO	NO
ACTION 2	NO	NO	YES	NO
ACTION 3	NO	NO	NO	YES

Table 8.1 is a limited decision entry table. It comprises of four areas called the condition stub, the condition entry, the action stub and the action entry. Each column of the table is a rule that specifies the condition under which the actions listed in the action stub will be taken. The condition stub is a list of names of conditions. A rule specifies whether a condition should or should not be met for the rule to be satisfied. "YES" means that the condition must be met, "NO" means that the condition must not be met, and "-" means that the condition does not have a role to play in the condition. The action stub names the actions the routine will take or initiate if the rule is satisfied. If the action entry is "YES," the action will be taken; if "NO," the action will not be taken. Table 8.1 can be translated as under:

1a. Action 1 will be taken if the condition 1 and 4 are satisfied and if conditions 2 and 3 are not met (rule1), or condition 2 and 4 are met and condition 1 and 3 are not met (rule2).

Condition is an alternate word for predicate: either a predicate in a specification or c control-flow predicate in a program. A condition is satisfied means that the predicate is true. Similarly, for "not met" and "false".

In addition to the actions specified we need to mention the default action that needs to be taken when all the other rules fail. Default rules for Table 8.1 are as specified in Table 8.2.

Table 8.2: Default Rules for Table 8.1

	RULE 5	RULE 6	RULE 7	RULE 8
CONDITION 1	YES	NO	NO	NO
CONDITION 2	NO	YES	NO	-
CONDITION 3	NO	NO	NO	-
CONDITION 4	YES	YES	NO	YES
DEFAULT ACTION	YES	YES	YES	YES

If the set of rules cover all possible combinations of TRUE/FALSE for the predicates, a default specification is not required.

8.3.2 Decision Table Processors

Decision tables can be translated into code and are a high-order language. The decision table's translator checks the source decision table for consistency and completeness and fills in any required default rules. Decision tables as a source language have the virtue of clarity, direct correspondence to specifications and maintainability. The main deficiency is the possible object-code inefficiency. They prove a useful tool for business data processing.

8.3.3 Decision Tables as a Basis for Test Case Design

If the specification is given as a decision table, the decision table must be used for test case design. Before doing so, the consistency and completeness of the decision table must be checked by the decision-table processor; therefore, it would seem that there would be no need to design those test cases. It is true that the testing is not needed to expose contradictions and inconsistencies but to determine whether the rules themselves are correct and to expose possible bugs in processing on which the rules' predicates depend.

It is not always possible or desirable to implement the program as a decision table because the program's logical behavior is only a part of the behavior of the decision table which specifies the program's logic. Decision table may not have the necessary features to feature the program's interfaces with other programs. Thus the use of decision tables to design tests is good when:

1. The specification is given as a decision table or can be converted to one easily.
2. The order in which the predicates are evaluated does not affect interpretation of the rules or resulting action.
3. The order of evaluation of rules does not affect the resulting action.
4. Once a rule is satisfied and an action is selected, no other rules need to be examined.
5. If several actions can result from satisfying a rule, the order in which the actions are executed doesn't matter.

The above restrictions have further implications: (1) the rules are complete in the sense that every combination of predicate truth values, including the default cases, are inherent in the decision table, and (2) the rules are consistent if and only if every combination of predicate truth values result in only one action or set of actions. If the set of rules are incomplete, there could be a combination of inputs for which no action, normal or default, are specified and the routine's action would be unpredictable.

8.3.4 Expansion of Immaterial Classes

Decision table contradictions are caused by the improperly specified immaterial entries (-). The key to test case design based on decision tables is to expand the immaterial entries and to generate tests that correspond to all the subrules that result. If some conditions are three-way, an immaterial entry expands into three subrules. Similarly, an immaterial n-way case statement expands into n subrules.

Expansion of the immaterial subcases in Table 8.2 is as given in Table 8.3.

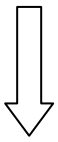
Table 8.3: Expansion of Immaterial Cases for Rule 4

	RULE 4.1	RULE 4.2	RULE 4.3	RULE 4.4
CONDITION 1	NO	NO	NO	NO
CONDITION 2	NO	NO	YES	YES
CONDITION 3	NO	YES	NO	YES
CONDITION 4	YES	YES	YES	YES

If no default rules are given, then all cases not covered by explicit rules are perforce default rules. If default rules are given, then we must test the specification for consistency. The specification is said to be complete if and only if n (binary) conditions expand into exactly 2^n unique subrules. It is consistent if and only if all rules expand into subrules whose condition combinations do not match those of any other rules. Table 8.4 is an example of an inconsistent specification in which the expansion of two rules yields a contradiction.

Table 8.4: The Expansion of an Inconsistent Specification

	RULE 1	RULE 2
CONDITION 1	YES	NO
CONDITION 2	-	YES
CONDITION 3	NO	-
CONDITION 4	YES	YES
ACTION 1	YES	NO
ACTION 2	NO	YES



	RULE 1.1	RULE 1.1	RULE 2.1	RULE 2.1
CONDITION 1	YES	YES	NO	NO
CONDITION 2	NO	YES	YES	YES
CONDITION 3	NO	NO	NO	YES
CONDITION 4	YES	YES	YES	YES
ACTION 1	YES	YES	NO	NO
ACTION 2	NO	NO	YES	YES

8.3.5 Test Case Design

Test case design using decision tables begins with examining the specification's consistency and completeness. This is done by expanding all immaterial cases and checking the expanded tables. Also, make the default case explicit and treat it as just another set of rules for the default action. Once we have verified the specification, the

objective of the test cases is to show that the implementation provides the correct action for all combinations of predicate values.

1. If there are k rules over n binary predicates, there are at least k cases to consider and at most 2^n cases. Find input values to force each case.
2. Implement test cases by changing the orders of the predicates evaluation if possible, by changing the input values order. Try all the pairs of interchanges for a representative set of values.
3. If the implementation allows the rule evaluation order to be modified, test different orders for the rules by pair wise interchanges. One pair of predicate values per rule should be sufficient.
4. Identify the places in the routine where rules are invoked or where the processors that evaluate the rules are called. Identify the places where actions are initiated. Instrument all paths from the rule processors to the actions so that you can show that the correct action was invoked for each rule.

8.3.6 Decision Tables and Structure

Decision tables can be used to examine a program's structure. Figure 8.1 shows a program segment that consists of a decision tree. These decisions can lead to actions 1, 2 or 3 in various combinations. The decision table for the program in Figure 8.1 is Table 8.5. We have given sixteen cases in Table 8.6 which is the expansion of the immaterial cases of Table 8.5.

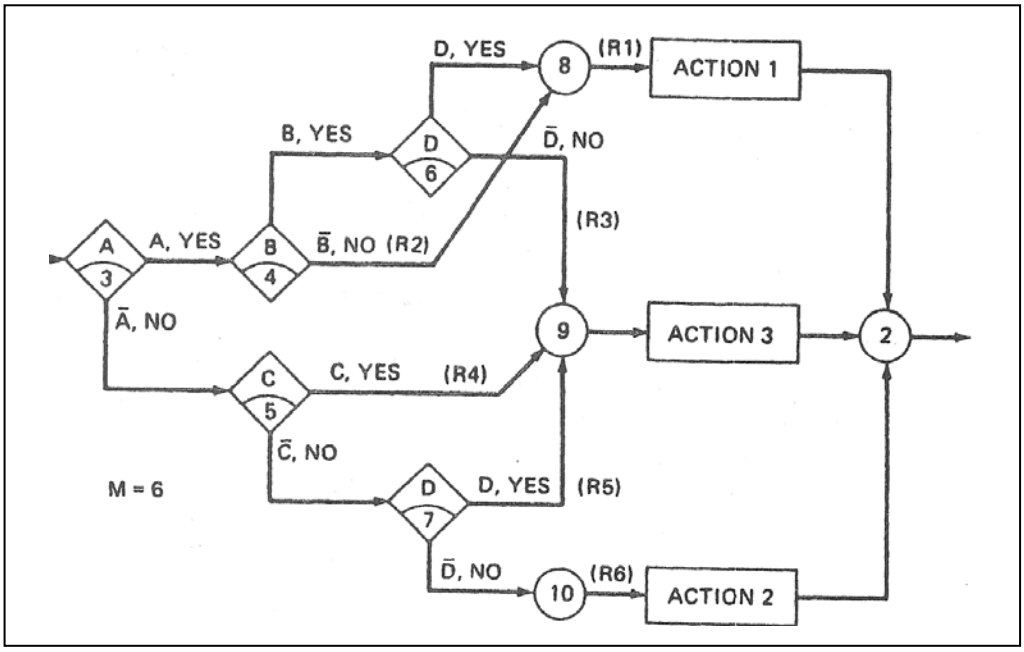


Figure 8.1: A Sample Program

Table 8.5: The Decision Table Corresponding to Figure 8.1

	RULE 1	RULE 2	RULE 3	RULE 4	RULE 5	RULE 6
CONDITION A	YES	YES	YES	NO	NO	NO
CONDITION B	YES	NO	YES	-	-	-
CONDITION C	-	-	-	YES	NO	NO
CONDITION D	YES	-	NO	-	YES	NO
ACTION 1	YES	YES	NO	NO	NO	NO
ACTION 2	NO	NO	YES	YES	YES	NO
ACTION 3	NO	NO	NO	NO	NO	YES

Table 8.6: Expansion of Table 8.5

	RULE 1	RULE 2	RULE 3	RULE 4	RULE 5	RULE 6
CONDITION A	Y Y	Y Y Y Y	Y Y	N N N N	N N	N N
CONDITION B	Y Y	N N N N	Y Y	Y Y N N	N Y	Y N
CONDITION C	Y N	N N Y Y	Y N	Y Y Y Y	N N	N N
CONDITION D	Y Y	Y N N Y	N N	N Y Y N	Y Y	N N

Check Your Progress 1

Fill in the blanks:

1. The user data is processed to obtain conclusions, using .....
2. Contradictions in the decision tables are caused .....
3. The specification is said to be complete .....

8.4 PATH EXPRESSIONS

Logic based testing is structural if it is applied to structure and functional when it's applied to a specification. In this type of testing we focus on the truth values of control flow predicates.

Predicates and Relational Operators

A predicate is implemented as a process whose outcome is a truth-functional value. Predicates are based on relational operators out of which the arithmetic operators are the most common ones. It is always better to look at the predicates from top-to-down from the point of view of the predicates as specified in requirements rather than from the point of view of predicates as implemented.

Case Statements and Multi-Valued Logics

Predicates need not be binary truth values alone. They could be multi-way predicates.

Shortcomings of Predicates

There are several things that can go wrong with predicates, especially if it has to be interpreted in order to express it as a predicate over input values.

1. The wrong relational operator is used.
2. The predicate expression of a compound predicate is incorrect.
3. The wrong operands are used.
4. The process leading to the predicate along its interpretation path is faulty.

8.4.1 Boolean Algebra

Notation

Let us now discuss the steps needed to be taken to get the predicate expression of a path.

1. Label each decision with a capital letter representing the truth value of the predicate. The YES/TRUE branch is named with a letter and the NO/FLASE branch with same letter overscored.
2. The truth value of a path is the product of he individual labels on it. Concatenation or product means "AND".

3. If two or more paths merge at a node, the fact is expressed using a plus (+) sign which stands for “OR”.

Using this convention, the truth-functional values for several of the nodes can be expressed in terms of segments from previous nodes. We use the node names to identify the point.

$N6 = \bar{A} + \bar{A}\bar{B}\bar{C}$

$N8 = (N6)B + \bar{A}B = AB + \bar{A}\bar{B}\bar{C}B + \bar{A}B$

The “OR” in boolean algebra is always an inclusive OR, which means “A or B or both”. The exclusive OR, which means “A or B, but not both” is $\bar{A}B + A\bar{B}$.

There are only two numbers in Boolean algebra: zero (0) and one (1). One means “always true” and zero means “always false”.

Rules of Boolean Algebra

X meaning AND. Also called multiplication. A statement such as AB means “A and B are both true.”

\+ meaning OR. “A + B” means “either A is true or B is true or both.”

$\bar{A}$ meaning NOT. Also negation or complementation. This is read as either “not A” or “A bar”. The entire expression under the bar gets negated.

Let us now have a look at the laws of Boolean algebra:

$A \bullet B = B \bullet A$		
$A + B = B + A$		Commutativity of AND(•), OR(+)
$A \bullet (B \bullet C) = (A \bullet B) \bullet C$		
$A + (B + C) = (A + B) + C$		Associativity of •, +
$A = B \bullet B = C \Rightarrow A = C$ (not Eq law)		Transitivity of “=”
$A \bullet (B + C) = (A \bullet B) + (A \bullet C)$		Distributivity of • over + reverse direction = Factoring out •
$A + (B \bullet C) = (A + B) \bullet (A + C)$		Distributivity of + over • reverse direction = Factoring out +
$1 \bullet A = A$		
$0 + A = A$		Identity element, Unit value "1 is the identity "0 is the identity (unit value) for AND" (unit value) for OR"
$0 \bullet A = 0$		
$1 + A = 1$		Mask "0 masks values 1 masks values (prevents from getting (prevents from getting through) AND, through) OR"
$\bar{A} \bullet A = 0$	$\bar{A} + A = 1$	Inverse element, Cancellation
$A \bullet A = A$		
$A + A = A$		Simplification, Duplication

$A \bullet B \Rightarrow A$		(not Eq laws)
$A = F \Rightarrow (A \bullet B) = F$		Absorbtion, negative Expansion of •
$A \Rightarrow A + B$		(not Eq laws)
$A + B = F \Rightarrow A = F$		Expansion, negative Absorbtion of +
$\overline{A \cdot B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \cdot \overline{B}$	DeMorgan's Laws
		Negation distributes while changing operation reverse direction = Negation moves outward

Individual letters in a Boolean algebra expression are called literals. The product of several literals is called a product term. Product terms can be simplified by removing duplicate appearances of the same literal barred or unbarred. E.g. AAB can be replaced by AB.

And $A\,\overline{B}\,C\,\overline{B}$ can be replaced by $A\,\overline{B}\,C$. Also, any product term that contains both the barred and unbarred appearance of the same literal can be removed because it is equivalent to 0 from the rule Inverse element or cancellation given above. Any Boolean expression that has been multiplied out so that it consists of the sum of products (e.g. AB + CDE+ FG) is said to be in the sum-of-products from.

Examples:

$N6 = A + \overline{A}\,\overline{B}\,\overline{C}$

$= A + \overline{B}\,\overline{C}$

$N8 = (N6)B + \overline{A}\,B$

$= (A + \overline{BC})\,B + \overline{A}\,B$

$= (AB + \overline{BC}\,B + \overline{A}\,B)$

$= (AB + B\,\overline{BC} + \overline{A}\,B)$

$= (AB + 0C + \overline{A}\,B)$

$= (AB + \overline{A}\,B)$

$= (A + \overline{A})\,B$

$= 1 \times B$

$= B$

Paths and Domains

A product term on an entry/exit path specifies a domain because each of the underlying predicate expression specifies a domain boundary over the input space. In case of compounded predicates, the Boolean path would be a sum of product terms like AB + CDE + FG. Because this expression was derived from one path, the expression also specifies a domain. However, this domain not be connected simply and each of the three terms could correspond to three separate disconnected subdomains. If any of the product term is included in the other product term, e.g. in ABC + AB, ABC is contained in AB, so we can always get rid of ABC through Boolean algebra simplification.

An alternate approach could have eliminated the compound predicates by providing a separate path for each product term. Example, we can implement AB + CDE + FG can be as implemented as one path using a compound predicate or as three different paths (AB, CDE, FG) specifying three separate domains that call the same processing subroutine to calculate the outcomes.

Let's say that we've written our program, design or specification such that there is only one product term for each domain: call these D_1, D_2, …, D_i, … , D_m. Consider any two of these product terms, say D_iD_j. For every i not equal to j, the product terms, D_i and D_j must be equal to zero. If the product is not equal to zero, then the domain are overlapping, i.e. the domain specification is contradictory. Also, the sum of all D_i must be equal to 1 otherwise there is an ambiguity. The same will hold even if D_i is not a simple product term but arbitrary Boolean expression resulting from compound predicates i.e. the domains are not simply connected.

Test Case Design

In principle, it is necessary to design a test case for each possible combination of TRUE/FALSE of the predicates. If all the predicates are uncorrelated, then each of the 2^n combinations would correspond to a different path, a different domain and their sum would correspond to a minimum covering set.

We can have contradictions in a specification but not in a program, if:

1. The routine has a single entry and a single exit.
2. No combination of predicate values leads to non-terminating loops.
3. There are no pieces of dangling code that leads nowhere.

Under these cases, the Boolean expression would be just 1. If it is not we have made a mistake in evaluation or there are pieces of unreachable code or non-terminating code for some predicate value combination.

The most complete set of test cases is the one which includes all the possible 2^n combinations of predicate values. This is an exhaustive case but testing can be achieved using lesser test cases.

Typically, we are interested in the Boolean algebra expression corresponding to paths from one node to another. In case of multiple nodes, we form a product of these simplified nodes to generate the test cases. Any one prime implicant in the Boolean expression covering all the paths from entry to the node is sufficient to reach the node. The set of paths used to reach any point of interest in the flow-graph can be characterized by an increasingly more thorough set of test cases:

1. ***Simples:*** Use any prime implicant in the expression to the point of interest as a basis for a path. The only that must be taken are those that appear in the prime implicant. All predicates not appearing in the prime implicant chosen can be set arbitrarily.
2. ***Prime Implicant Cover:*** Pick input values so that there is at least one path for each prime implicant at the node of interest.
3. ***All terms:*** Test all expanded terms for that node.
4. ***Path dependence:*** The truth value of a predicate depends upon the path taken to reach that predicate.

8.4.2 Boolean Equations

Loops can be complicating as we may have to solve a Boolean equation to determine which predicate value combinations lead us where. Also, the Boolean expression for the end point does not necessarily equal 1. Consider the flow-graph of Figure 8.2.

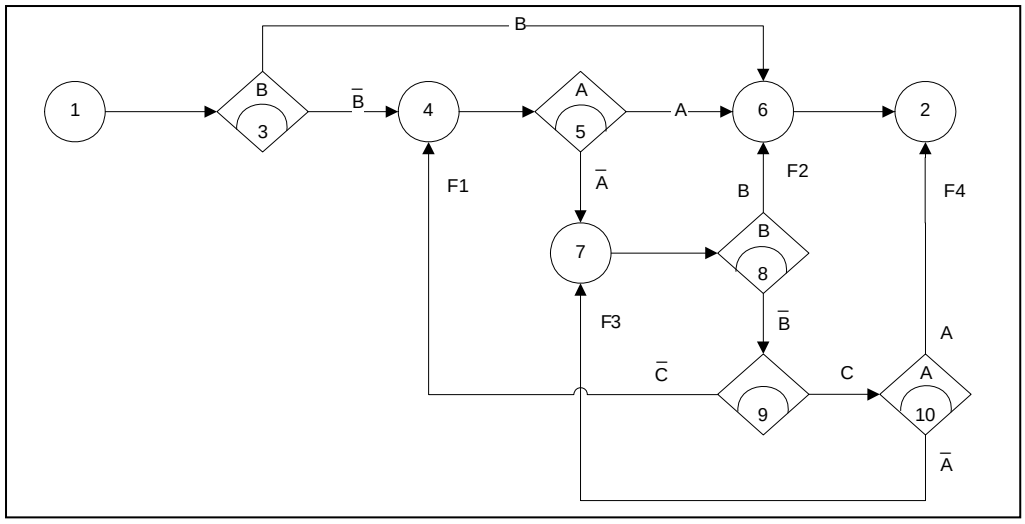


Figure 8.2: A Flow Graph with Loops

From this graph we have,

$N4 = \bar{B} + F1$

$F1 = \bar{B}\,\bar{C}\,N7$

$N4 = \bar{B} + \bar{B}\,\bar{C}\,N7$

$= \bar{B}$

$N6 = B + AN4$

$= B + \bar{A}\,B$

$= A + B$

$N7 = \bar{A}\,N4 + F3$

$= \bar{A}\,\bar{B} + F3$

$F3 = N7\,\bar{B}\,C\,\bar{A}$

$N7 = \bar{A}\,\bar{B} + \bar{A}\,\bar{B}\,C\,N7$

$= \bar{A}\,\bar{B}$

$N2 = N6 + F4$

$= A + B + F4$

$F4 = A\,\bar{B}\,CN7$

$N2 = A + B + A\,\bar{B}\,CN7$

$= A + B$

The fact that the exit expression is not equal to 1 implies that there are loops. Feeding the logic back into itself this way, usually in the interest of saving some code or some work, leads to simultaneous Boolean equations, which are rarely as easy to solve as the given example; it may also lead to dead paths and infinite loops.

Check Your Progress 2

1. Define literals.

   .....

   .....

2. (A + B).(C+D) = ...

3. A predicate could have a truth-functional value or multiple values. (True/False)

8.5 KV CHARTS

8.5.1 The Problem

While dealing with Boolean expressions involving more than three variables, the designing and generation of test cases using the Boolean algebraic express becomes difficult. The Karnaugh-Veitch chart reduces the Boolean algebraic manipulations to graphical trivia.

8.5.2 Simple Forms

Figure 8.3 shows all the Boolean functions of a single variable in the form of KV charts.

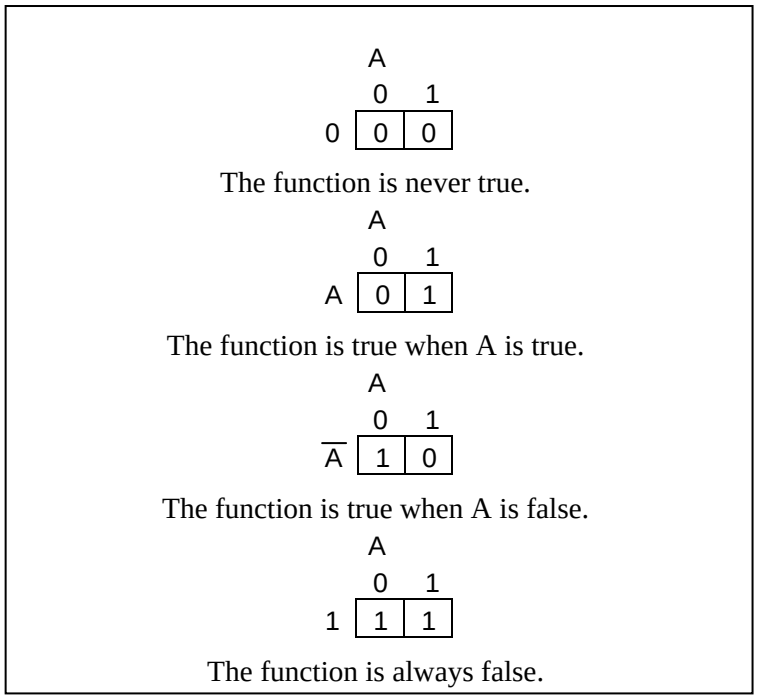


Figure 8.3: KV Charts for Functions of a Single Variable

These charts show all possible truth values that the variable A can have. The heading above each box denotes this fact. A “1” means the variable’s value is “1” or TRUE. A “0” means the variable’s value is “0” or FALSE. The entry in the box (0 or 1) specifies if the function that the chart represents is true or false for that value of the variable.

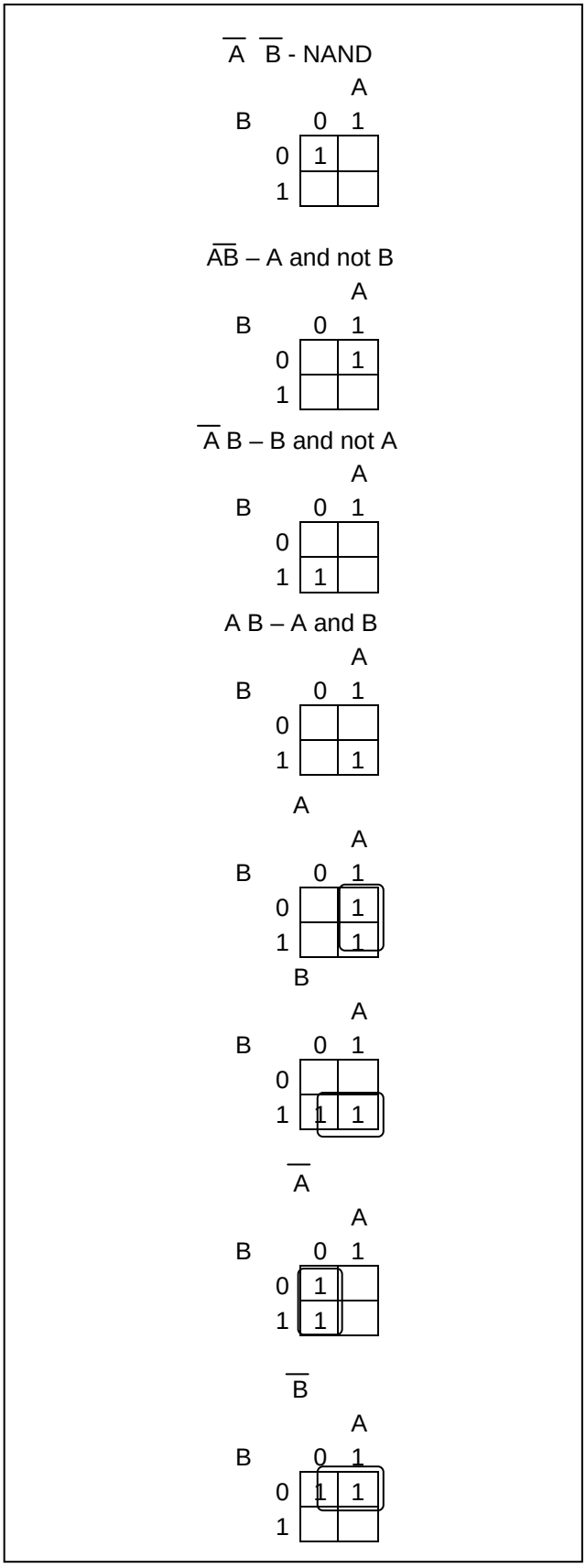


Figure 8.4: Functions of Two Variables

$\overline{A}\ \overline{B} + AB$

A If and only if B

B \ A	0	1
0	1	
1		1

$A\overline{B} + \overline{A}B$

Exclusive OR

B \ A	0	1
0		1
1	1	

A + B

B \ A	0	1
0		1
1	1	1

$\overline{A} + B$

A implies B

B \ A	0	1
0	1	
1	1	1

$\overline{A} + \overline{B}$

B \ A	0	1
0	1	1
1	1	

$\overline{B} + A$

B implies A

B \ A	0	1
0	1	1
1		1

Universal False

B \ A	0	1
0		
1		

Universal True

B \ A	0	1
0	1	1
1	1	1

Figure 8.5: More Functions of Two Variables

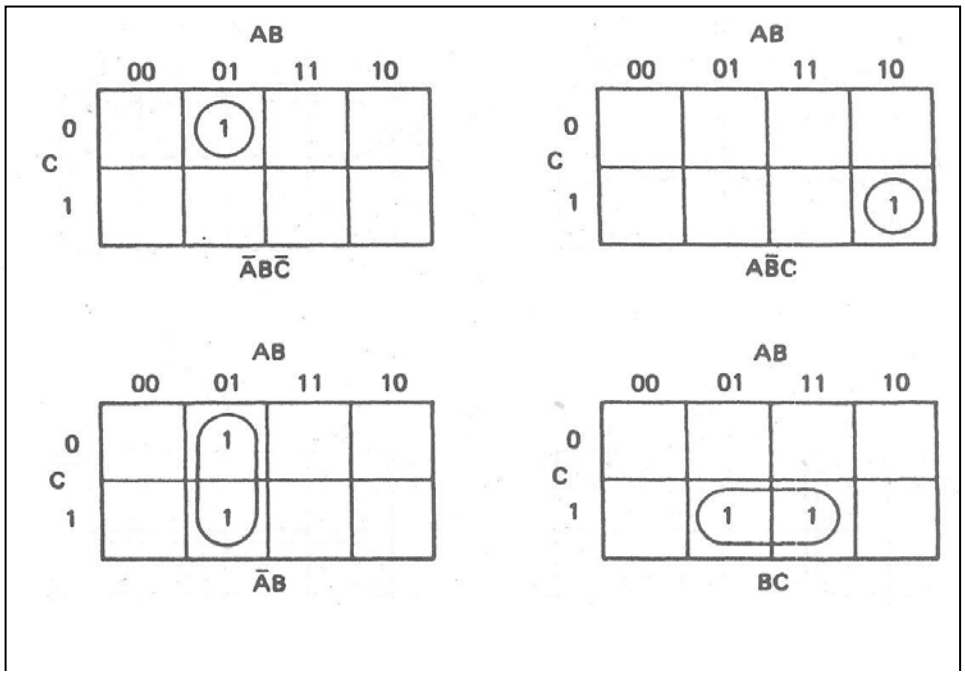
Figure 8.4 shows eight of the sixteen possible functions of two variables. Each box corresponds to the combination of values of the variables for the row and column of that box. Variables for the row and column of that box. The single entry for AB in the first chart is interpreted that way because both A and B value for the box is 0. Similarly, AB corresponds to A =1 and B = 0, A B to A = 0 and B = 1, and AB to A = 1 and B = 1. The next four functions have two non-zero entries each and each entry forms an adjacent pair either horizontally or vertically but not diagonally. Any variable that changes in either the horizontal or vertical direction does not appear in the expression.

Figure 8.5 shows the remaining eight functions of two variables. The interpretation of these charts is a combination of interpretation of previous charts.

Because KV charts are Boolean functions, two charts over the same variable, arranged in the same way, their product is the term-by-term product, their sum is the term-by-term sum, and the negation of a chart is obtained by reversing all the 0 and 1 entries in the chart. In order to simplify an expression using a KV chart, we fill each term one at a time, and then look for adjacencies and to rewrite the expression in terms of the largest grouping you can find that cover all the 1's in the chart.

8.5.3 Three Variables

KV charts for three variables are shown below. Here we have named the columns in an unusual way like "00, 01, 11, 10" rather than with the expected "00, 01, 10, 11". This is because this labeling preserves the adjacency properties of the chart. Adjacency can also go around the corners. Two boxes are said to be adjacent if they change in only one bit and two groupings are adjacent if they change in only one bit. A three variable KV chart can have groupings of 1, 2, 4 and 8 boxes as illustrated in Figure 8.6.



Contd...

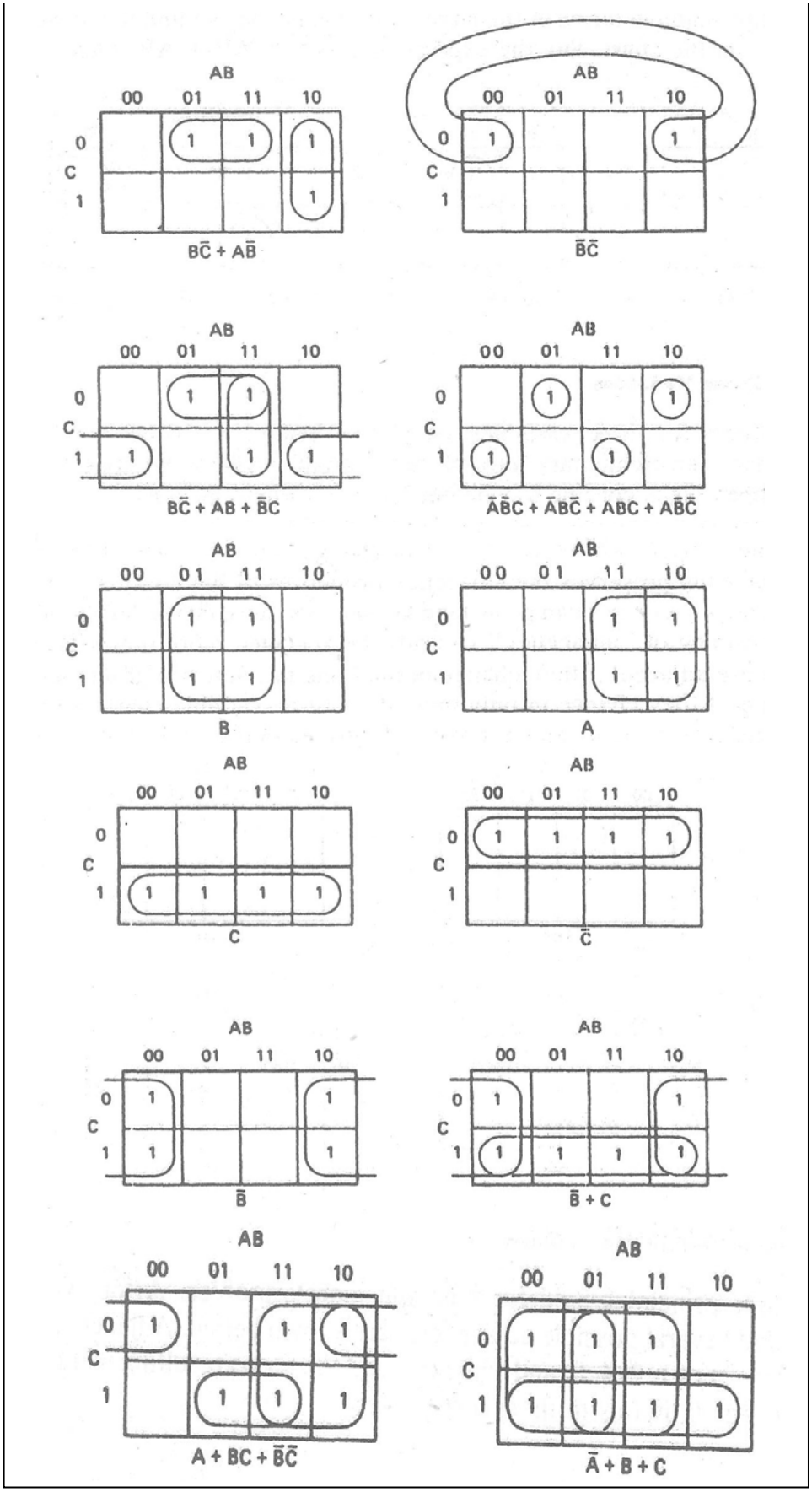


Figure 8.6: Three Variable Functions

8.5.4 Four Variables and More

The same principle holds for four and more variables. A four variable chart and its adjacencies are shown below:

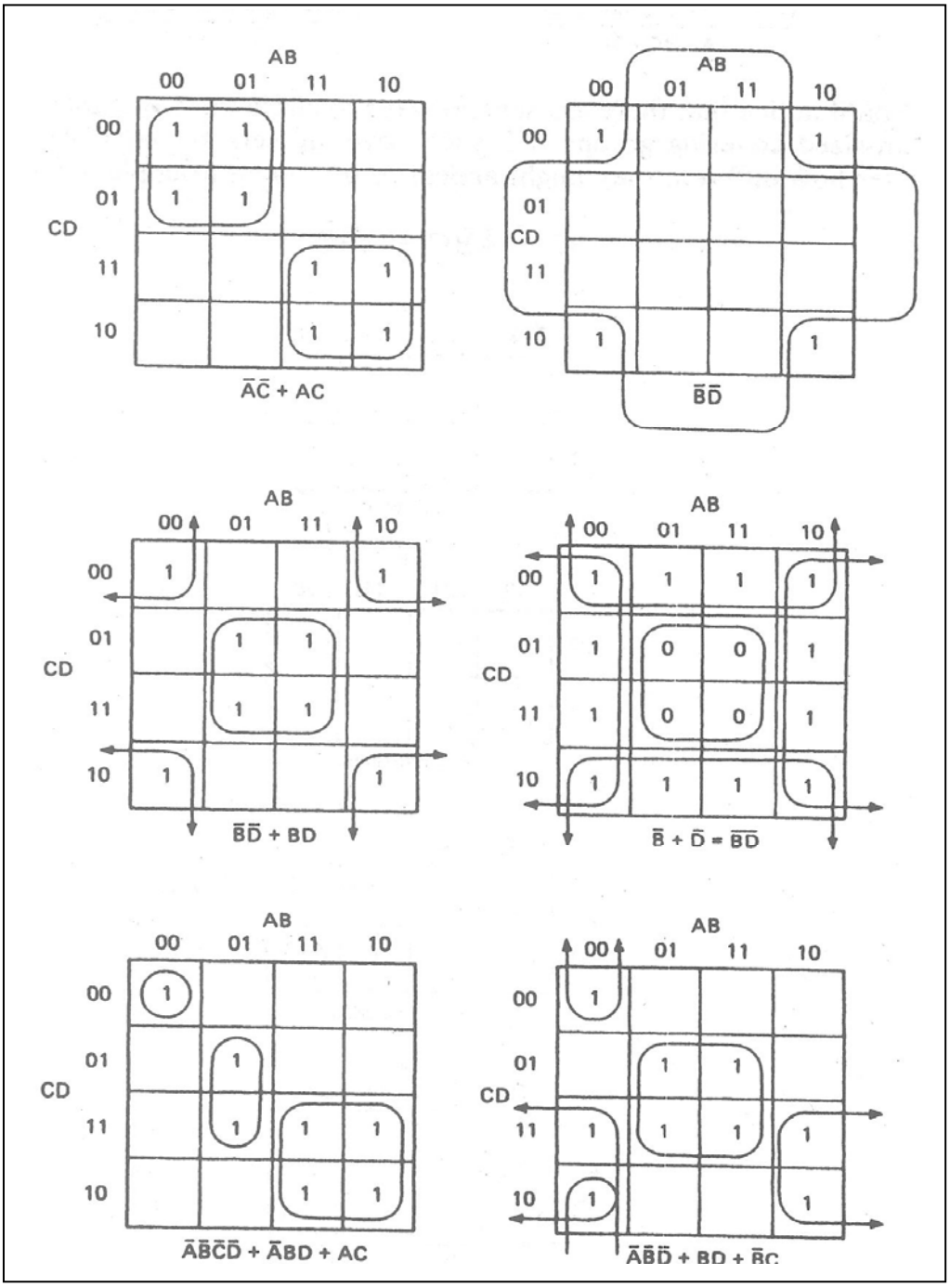


Figure 8.7: Four Variable Functions

8.6 SPECIFICATIONS

Specification should be logically consistent and complete otherwise design and coding is useless. The procedure for specification includes the following:

1. Rewrite the specification using consistent terminology.

2. Identify the predicates on which the cases are based. Name them with suitable letters like A, B and C.
3. Rewrite the specifications in English using logical connectives like AND, OR and NOT, as suitable.
4. Convert the rewritten specification into an equivalent set of Boolean expressions.
5. Identify the default action and cases, if any are specified.
6. Enter the Boolean expression in a KV chart and check for consistency. If the specifications are consistent, there will be no overlaps, except for the cases that result in multiple actions.
7. Enter the default cases and check for consistency.
8. If all boxes are covered, the specification is complete.
9. If the specification is incomplete or inconsistent, translate the corresponding boxes of KV chart back to English and get a clarification, explanation or revision.
10. If the default case were not specified explicitly, translate the default cases back to English and get a confirmation.

8.6.1 Finding and Translating Logic

We begin writing the specifications by getting rid of ambiguous terms, words and phrases and expressing it all as a long list of IF..THEN statements. We then identify the actions and name them as A1, A2, A3, etc. Break the actions into small units first. All actions at this point should be mutually exclusive in the sense that no one action is part of another. If some actions always occur in combination with other actions and vice versa, then lump them into one action and give it a new name. Now substitute the action names in the sentences. Identify the "OR" component of all sentences and rewrite them so that each "OR" is on a separate line.

This process should be done as early as possible because the translation of the specification into Boolean algebra may require discussion among the specifiers, especially if contradictions and ambiguities emerge. If the specification has been given as a decision table or in another equally unambiguous tabular form, then most of the above work has been avoided and so have much of the potential confusion and the bugs that inevitably result therefrom.

8.6.2 Ambiguities and Contradictions

Here is a specification:

$$A1 = \bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\,CD$$

$$A2 = A\,\bar{C}\,D + A\,C\,D + \bar{A}\bar{B}\,C + A\,\bar{B}\,C$$

$$A3 = B\,D + B\,\bar{C}\,D$$

$$ELSE = B\,C + \bar{A}\bar{B}\bar{C}\bar{D}$$

The KV chart for this specification is as under:

		AB			
		00	01	11	10
CD	00	4	1	1, 2	2
	01		3	2, 3	1, 2
	11	4	3	3	4
	10	4	3	3	4

There is an ambiguity, may be related to the default case: $\bar{A}\ \bar{B}\ \bar{C}\ D$ is missing. The specification layout seems to suggest that this term also belongs to the default action. Thus, we need to answer the following questions:

1. Is $\bar{A}\ \bar{B}\ \bar{C}\ D$ also to be a considered a default action?
2. Shall the default action be rephrased as $\bar{B}\ C + \bar{A}\ \bar{B}$?

There may be various boxes in a KV chart that call for multiple actions. If the specification does not mention these actions explicitly then these actions should be considered as conflict.

If no explicit default action has been specified, then fill the KV chart with explicit entries for the explicitly specified actions, negate the entire chart and present the equivalent expression as a statement of the default. We also need to be suspicious of nearly complete adjacencies. For example, if a term contains seven adjacent boxes and lacks only one to make a full eight adjacency, question the missing box.

It is also useful to represent the new specification as a table which shows all cases explicitly and also as a compact version in which you have taken advantage of the possibility of simplifying the expression by using a KV chart.

8.6.3 Don't-Care and Impossible Terms

There are two kinds of so-called impossible conditions: (1) the condition cannot be created or is seemingly contradictory or improbable; and (2) the condition results from our insistence on forcing a complex, continuous world into a binary, logical world. Most program illogical conditions are of the second kind. These so-called impossible cases help in simplifying the program the implements logic. We can take the advantage of an impossible case only when we are sure that there is a data validation or protection in a preceding module or when appropriate illogical condition checks are made elsewhere in the program. Although this practice is dangerous and must be avoided but even if you do it, you must do it right:

1. Identify and confirm all the "impossible "and "illogical" cases.
2. Fill the KV chart first with the possible cases and then with the impossible ones. Use the combined symbol 0 which can be interpreted as 0 or 1 depending upon the value which simplifies the logic the most. These terms are called don't-care terms, because the case if presumed impossible and we don't care which value is used.

8.7 LET US SUM UP

To organize statements in a specification, use decision tables as an intermediate step towards a more revealing equivalent Boolean algebra expression.

Label the links following binary decisions with a weight that corresponds to the predicate's logical value, and evaluate the Boolean expressions to the nodes of interest.

Simplify the resulting expressions or solve equations and then simplify if you cannot directly express the Boolean function for the node in terms of the path predicate values.

The Boolean expression for the exit node should equal 1. If it does not, or if attempting to solve for it leads to a loop of equations, then there are conditions under which the routine will loop indefinitely. The negation of the exit expression specifies all the combinations of predicate values that will lead to the loop or loops.

By deriving a test case from the expansion of any prime implicant in the Boolean expression for that node, we can reach any node of interest.

The set of all paths from the entry to a node can be obtained by expanding all the prime implicants of the Boolean expression that corresponds to that node. A branch-covering set of paths, however, may not require all the terms of the expansion.

Use KV charts for expressions involving up to six variables.

Be careful while translating English into Boolean algebra. Retranslate and discuss the retranslation of the algebra with the specifier.

Question all missing entries, question overlapped entries if there was no explicit statement of multiple actions, and question all almost-complete groups.

Don't take advantage of don't-care cases or impossible cases unless you are willing to pay the maintenance penalties; but if you must, get the maximum payoff by making the resulting logic as simple as you can and document all instances in which you take advantage.

8.8 LESSON END ACTIVITIES

1. How can we use decision tables to carry out logic-based testing?
2. Mention the three operators included in Boolean algebra.
3. What are Boolean equations? Simplify the following Boolean expressions.

   (a) $(A + C)(AD + A\ \bar{D}) + AC + C$

   (b) $\bar{A}(A + B) + (B + AA)(A + \bar{B})$

8.9 KEYWORDS

Adjacency: Two boxes are said to be adjacent if they change in only one bit and two groupings are adjacent if they change in only one bit.

Sum-of-products form: Any Boolean expression that has been multiplied out so that it consists of the sum of products (e.g. AB + CDE+ FG) is said to be in the sum-of-products from.

Literals: Individual letters in a Boolean algebra expression are called literals.

Product term: The product of several literals is called a product term.

8.10 QUESTIONS FOR DISCUSSION

1. What are the KV charts? What are their uses? Explain the various kinds of KV charts.
2. Simplify the expressions given in section 8.8 question (3) above using KV charts.
3. What are don't care terms and how can they be useful?

Check Your Progress: Model Answers

CYP 1

1. inference engine
2. immaterial entries (-)
3. if and only if n (binary) conditions expand into exactly 2^n unique subrules

Contd...

CYP 2

1. Individual letters in a Boolean algebra expression are called literals.
2. A.C + A.D + B.C + B.D
3. True

8.11 SUGGESTED READINGS

Boris Beizer, *Software Testing Techniques*, Second Edition, Dreamtech Press, 2003.

Myers and Glenford, J., *The Art of Software Testing*, John-Wiley & Sons, 1979.

Roger, S. Pressman, *Software Engineering – A Practitioner's Approach*, 5th Edition, McGraw Hill, 2001.

Marnie, L. Hutcheson, *Software Testing Fundamentals*, Wiley-India, 2007.

LESSON

9

STATES, STATE GRAPHS AND TRANSITION TESTING

CONTENTS

9.0 Aims and Objectives

9.1 Introduction

9.2 State Graphs

- 9.2.1 States
- 9.2.2 Inputs and Transitions
- 9.2.3 Outputs
- 9.2.4 State Tables
- 9.2.5 Time versus Sequence
- 9.2.6 Software Implementation

9.3 State Graphs: Good and Bad

- 9.3.1 State Bugs
- 9.3.2 Transition Bugs
- 9.3.3 Output Errors
- 9.3.4 Encoding Bugs

9.4 State Testing

- 9.4.1 Impact of Bugs
- 9.4.2 Principles
- 9.4.3 Limitations and Extensions
- 9.4.4 What to Model
- 9.4.5 Getting the Data

9.5 Let us Sum up

9.6 Lesson End Activities

9.7 Keywords

9.8 Questions for Discussion

9.9 Suggested Readings

9.0 AIMS AND OBJECTIVES

After studying this lesson, you would be able to understand:

- State Graphs – States, Inputs and Transitions, outputs, State tables, time versus sequences, software implementation

- Good state graphs and bad – state bugs, transition bugs, output errors, encoding bugs
- State testing – Bug impacts, principles, limitations and tools

9.1 INTRODUCTION

The state-graph and its associated state table are useful models for describing software behavior. The finite-state machine is a functional testing tool and testable design programming tool. It is as fundamental to software engineering as Boolean algebra. State testing is based on the use of these machine models for software structure, software behavior or specifications of software behavior. They can be implemented as a table driven software which can serve as a powerful design option. They are most commonly used by testers for functional testing like system testing.

9.2 STATE GRAPHS

9.2.1 States

State is a combination of circumstances or attributes belonging for the time being to a person or thing. A program that detects the character sequence “ZCZC” can be in any of these states:

1. Neither ZCZC nor any part of it has been detected.
2. Z has been detected.
3. ZC has been detected.
4. ZCZ has been detected.
5. ZCZC has been detected.

States are represented as nodes. States can be numbered or may be identified by words or whatever is convenient. Figure 9.1 shows a typical state graph.

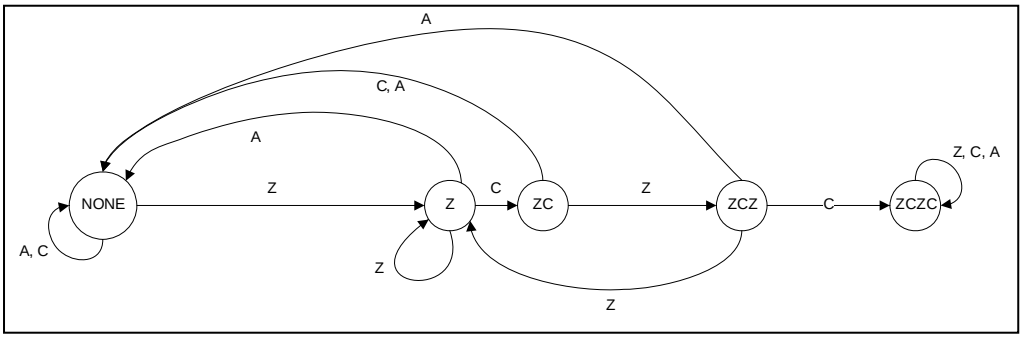


Figure 9.1: One-time ZCZC Sequence-Detector State Graph

The number of states in a computer is 2 raised to the power of number of bits in the computer; that is, all the bits in main memory, registers, discs, tapes and so on. State graphs are mostly useful for simpler functional models involving at the most a few dozen states and only a few factors.

9.2.2 Inputs and Transitions

State changes or transitions as a result of inputs. These transitions are represented as links joining the states. The input that causes the transition is marked on the link. There is an outlink from ever state for every input. If different inputs in a state cause a transition to the same state, instead of drawing a bunch of parallel links we can abbreviate the notation by listing the several inputs as in : “input1, input2, input3, …”.

A finite state machine is an abstract device that can be represented by a state graph having a finite number of states and a finite number of transitions between states.

The ZCZC detection example can have the following kinds of inputs:

1. Z
2. C
3. Any character other than Z or C, which we'll denote by A.

The state graph of Figure 9.1 can be interpreted as follows:

1. If the system is in the "NONE" state, any input other than a Z will keep it in that state.
2. If a Z is received, the system transitions to the "Z" state.
3. IF the system is in the "Z" state and a Z is received, it will remain in the "Z" state. If a C is received, it will go back to the "ZC" state; if any other character is receive, it will go back to the "NONE" state because the sequence has been broken.
4. A Z received in the "ZC" state progresses to the "ZCZ" state, but any other character breaks the sequence and causes a return to the "NONE" state.
5. A C received in the "ZCZ" state completes the sequence and the system enters the "ZCZC" state. A Z breaks the sequence and causes a transition back to the "Z" state; any other character causes a return to the "NONE" state.
6. The system stays in the "ZCZC" state no matter what is received.

Thus, a state-graph represents all this in a compact form.

9.2.3 Outputs

An output can be associated with any link. Outputs are denoted by letters or words and separated from inputs by a slash as follows: "input/output". Outputs are also link weights. If every input associated with a transition causes the same output, then denote it as: "input1, input2, …, input3/output." If there are many different combinations of inputs and outputs, it's best to draw a separate parallel link for each output. Let us now have a look at the tape control recovery routine state graph given in Figure 9.2.

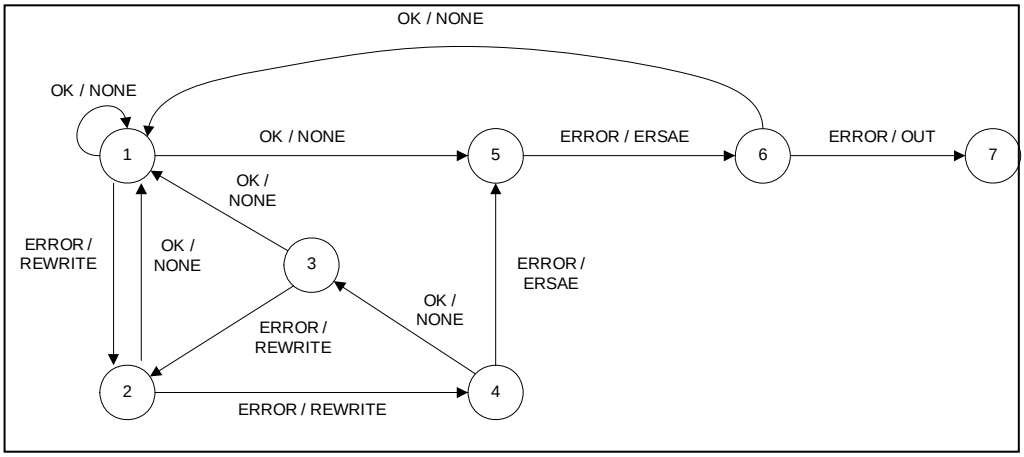


Figure 9.2: Tape Control Recovery Routine State Graph

The state graph shown in Figure 9.2 there are only two kinds of inputs (OK, ERROR) and four kinds of outputs (REWRITE, ERASE, NONE, OUT-OF-SERVICE).

9.2.4 State Tables

As big state graphs are cluttered and difficult to follow, its more convenient to represent the state graph as a table (the state table or state-transition table), specifies the states, inputs, transitions and outputs. The following conventions are used:

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

9.2.5 Time versus Sequence

State graphs do not represent time but only sequence. A transition may take microseconds or centuries; a system could be in one state for milliseconds and another for eons, or vice versa; the state graph would be the same because it does not include the time factor.

9.2.6 Software Implementation

There is hardly any direct correspondence between a program and a process described by a state graph. The state graph represents the total behavior consisting of the transport, software, executive, status returns, interrupts, etc. There is no simple correspondence between lines of code and states. The state table forms the basis for a widely used implementation. Four tables are involved:

1. A table or process that converts the input values to a compact list (INPUT_CODE_TABLE).
2. A table that specifies the next state for every combination of state and input code (TRANSITION_TABLE).
3. A table or case statement that specifies the output or output code, if any, associated with every state-input combination (OUTPUT_TABLE).
4. A table that stores the present state of very device or process that uses the same state table – e.g., one entry per tape transport (DEVICE_TABLE).

The routine operates in the following manner:

1. Fetch the present state from memory.
2. Fetch the present input value from memory. If it is numerical, use it directly otherwise encode it to a numeric value.
3. Combine the present state and the input code to obtain a pointer of the transition table and its logical image.
4. The output table contains a pointer to the routine to be executed for that state-input combination. The routine is invoked.
5. The same pointer is used to fetch the new state value which is then stored.

Input Encoding and Input Alphabet

Only the simplest finite-state machines can use the inputs directly. We may not be interested in using these input characters directly but in some attributes represented by the characters, e.g. in the ZCZC detector, we are only interested in three different types of inputs i.e. “Z”, “C” and “OTHER” and not all the 256v possible ASCII characters.

The alternative to input encoding is a huge state graph and tables because there must be one outlink in every state for every possible different input. Input encoding compresses the cases and therefore the state graph. Another advantage is that we can run the machine from a mixture of otherwise incompatible input events. The set of different encoded input values is called the input alphabet.

Output Encoding and Output Alphabet

There can be different types of incompatible outputs for transitions of a finite sate machine. Very rarely do we have a single character output for a finite state machine. We might want to get a string as output, call a subroutine, transfer control to a low-level finite-state machine or do nothing. Thus, we only have limited distinct actions that can be taken using which we can encode into a convenient output alphabet.

State Codes and State-Symbol Products

In a finite state machine, if there are n states and k different inputs, both numbered from zero, and the state code and input code are S and I respectively, then the pointer value is Sk+ I or In + S depending upon how we want to organize the tables. Finite-state machines are used in time critical applications because they have lower response times. A faster implementation is to use a binary number of states and a binary number of input codes and to form a pointer by concatenating the state and input code. The speed comes at the cost of some other disadvantages. The table no longer remains compact. The other disadvantage is size as the excessive table size could be a problem. For these reasons, there is another encoding of the combination of the state number and the input code into pointer. The term state-symbol product is used to mean the value obtained by any scheme used to convert the combined state and input code into a pointer to a compact table. Talking about “states” and “stat-codes” in the context of finite state machines we refer to hypothetical integer used to denote the state and not the actual form of the state code that could be a result of an encoding process. Similarly, “state-symbol product” means the hypothetical or actual concatenation used to combine the state and input codes.

Application Comments for Designers

An explicit state-table implementation is advantageous when either the control functionality is likely to change in future or when the system has many similar control functions which are slightly different. Such type of implementation is more common in telecommunications. This technique can provide fast response time, however, is not effective for very small or big state graphs.

Application Comments for Testers

Testers are concerned only with the way to design test cases using state graphs or state-table representations and not with the implementation of the approach. Also, at times the tests developed from a state-graph can help the developers to re-consider their implementation technique.

Check Your Progress 1

Fill in the blanks:

1. The set of different encoded input values is called the .....
2. The box at the intersection of row and column in a state table specifies the ..... and the .....

9.3 STATE GRAPHS: GOOD AND BAD

A state graph can be judged as good or bad depending upon these facts:

1. The total number of states is equal to the product of the possibilities of factors that make up the state.
2. For every state and input there is exactly one transition specifies to exactly one, possibly the same, state.
3. For every transition there is one output action specified. That output could be trivial, but at least one output does something sensible.
4. For every state there is a sequence of inputs that will drive the system back to the same state.

Figure 9.3 shows the examples of some improper state-graphs.

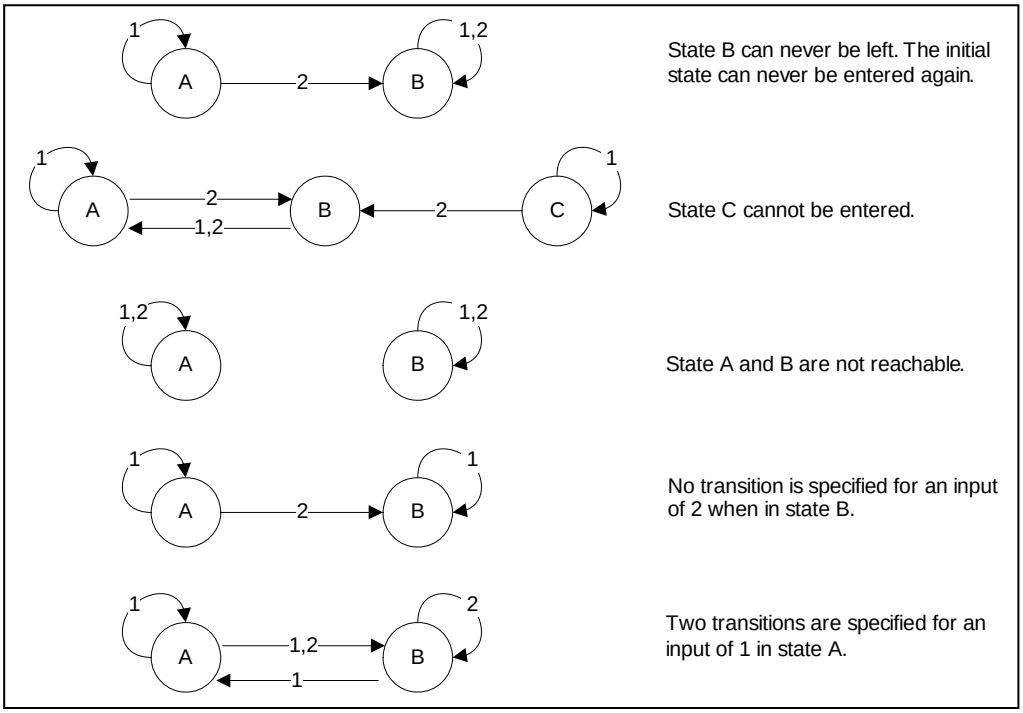


Figure 9.3: Improper State Graphs

A state graph must have at least two different input codes. With one input code, there are only a few kinds of state-graphs we can build: a bunch of disconnected individual states; disconnected strings of states the end in loops and their variations; or a strongly connected state graph in which all states are arranged in one grand loop. The latter can be implemented easily using a counter that resets at some specific values and thus, does not need an elaborate modeling apparatus.

In state testing we ignore outputs because the states and transitions are of primary importance to us. Two state graphs with identical states, inputs and transitions can have vastly different outputs, yet they could be identical from the point of view of state-testing.

9.3.1 State Bugs

Number of States

The number of states in a state graph is the number of states we choose to recognize or model. The state is recorded as a combination of values of variables that appear in the

database. Failure to account for all the states is one of the most common type of bug that can be modeled using a state-graph. Because an explicit state-table mechanism is not typical, the opportunities for missing states exist. We can find the number of states as follows:

1. Identify all the component factor of the state.
2. Identify all the allowable values for each factor.
3. The number of states is the product of the number of allowable values of all the factors.

Impossible States

Some combinations of factors appear to be impossible. The difference between a programmer's state count and the tester's state count is mainly due to a difference in opinion of the concerning "impossible states".

The "impossible" states can occur as the states we deal with inside computers are not the states of the real world but rather a numerical representation of those.

Equivalent States

Two states are said to be equivalent if every sequence of inputs starting from one state produces exactly the same sequence of outputs when started from the other state. Figure 9.4 depicts the situation.

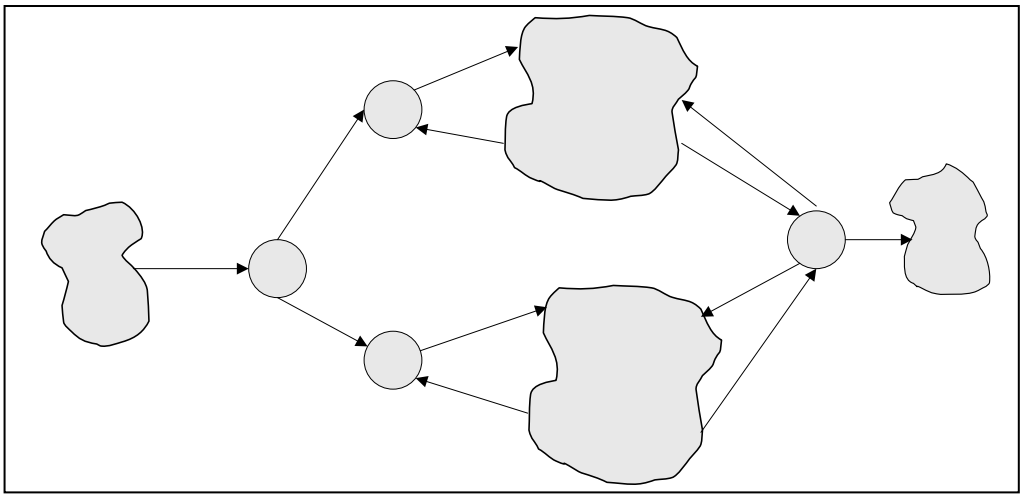


Figure 9.4: Equivalent States

Let us assume that a system is in state S, an input a causes a transition to state A and an input b causes a transition to state B. If starting from state A, every possible sequence of inputs produces exactly the same sequence of outputs that would occur when starting from B, then there is no way for an outsider to determine in which of the two states, the system is in, without observing the record of the states. The state graph can be reduced to that of Figure 9.5.

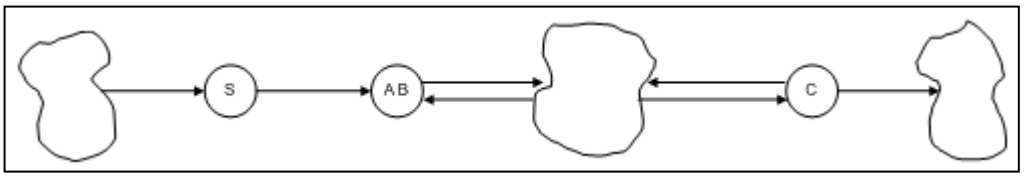


Figure 9.5: Equivalent States of Figure 9.4 Merged

Equivalent states can be identified using the following steps:

1. The rows corresponding to the two states are identical w.r.t. input/output/next state but the name of the next state could differ. The two states are differentiated only by the input as in Figure 9.6.
2. There are two sets of rows which, except for the state names, have identical state graphs w.r.t. transitions and outputs. The two sets can be merged as in Figure 9.7.

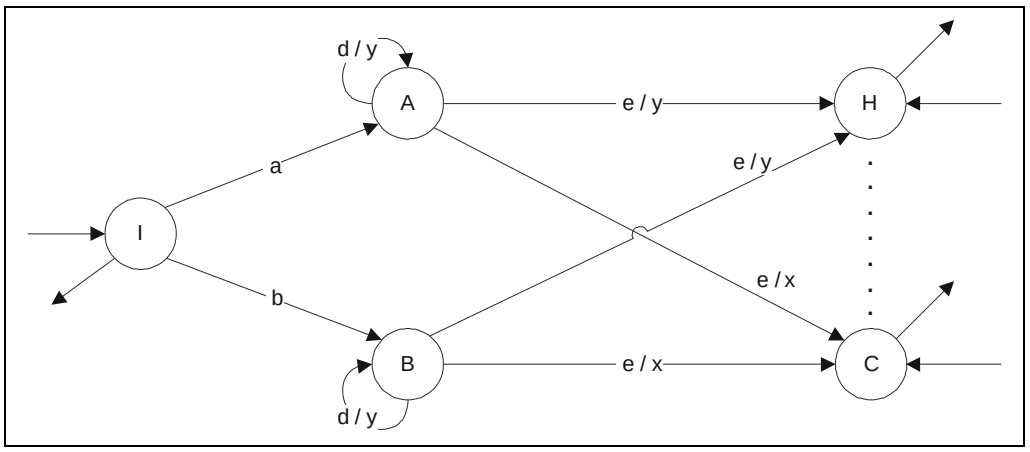
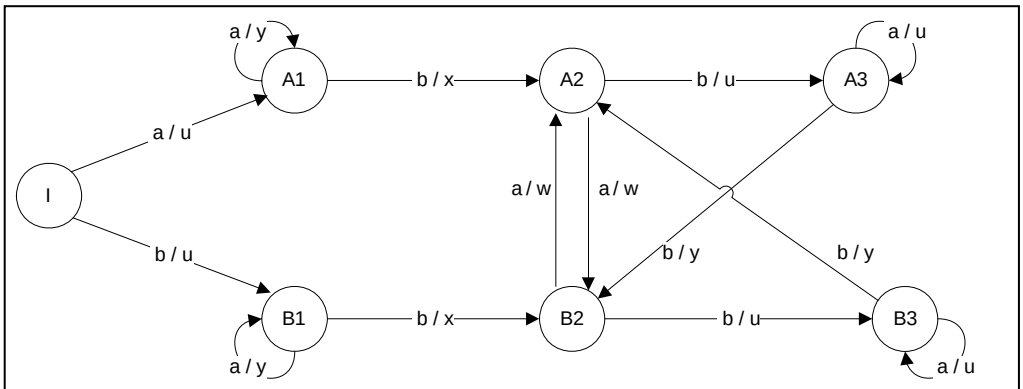


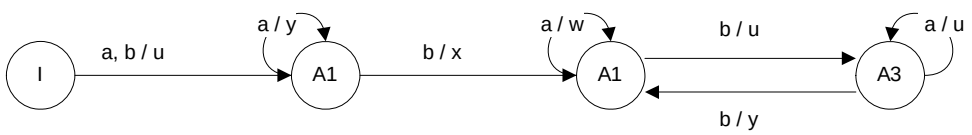
Figure 9.6: Equivalent States



(a)

STATE	A	B
I	A1/Y	B1/U
A1	A1/Y	A2/X
B1	B1/Y	B2/X
A2	B2/W	A3/U
B2	A2/W	B3/U
A3	A3/U	B2/Y
B3	B3/U	A2/Y

(b)



(c)

Figure 9.7: Merged Equivalent States

Bugs result due to unjustifiable merger of seemingly equivalent states. The two states or two set of states appear to be equivalent because the programmer has failed to carry through to a proof of equivalence for every input sequence.

9.3.2 Transition Bugs

Unspecified and Contradictory Transitions

Every input-state combination must have a specified transition. If not possible, then there must be a mechanism that prevents the input from occurring in that state. The transition may not be specified because of an oversight. There must be exactly one transition for every combination of input and state.

A system cannot have contradictions or ambiguities because the program will do something for every input irrespective if it is right or wrong. Even if the state does not change, it is considered as a transition to the same state.

Unreachable States

An unreachable state is the one that cannot be reached. It is not impossible. There may be transitions from an unreachable state to other states because the state became unreachable itself because of some wrong transitions.

An isolated unreachable state here and there, which relates to impossible combinations of real-world state-determining conditions, is acceptable, but if you find groups of connected states that are isolated from others, there's cause for concern. There are two possibilities: (1) there is a bug i.e. some transitions are missing. (2) The transitions are there, but you don't know about it, i.e. there are other inputs and associated transitions to reckon with. Such hidden transitions are caused by software operating at a higher priority level or by interrupting processing.

Dead States

A dead state (or a set of dead states) is a state that once entered cannot be left. It may not necessarily be a bug. A set of states may appear to be dead because the program has two modes of operation. In the first mode it goes through an initialization process that consists of several states. After this, it goes to a strongly connected set of working states, which, within the context of the routine, cannot be exited. The initialization states are unreachable to the working states, and the working states ate dead to the initialization states. Legitimates dead states are rare. They occur during system-level issues and device handlers.

9.3.3 Output Errors

Output could be incorrect, although inputs, states and transitions are correct and there are no dead and unreachable states. Output actions must be verified independent of states and transitions. The most likely reason for an incorrect output could be an incorrect call to the routine that executes the output. This is a minor bug. Bugs in the state graphs are more serious because they tend to be related to fundamental control-structure problems. If the routine is implemented as a state table, both types of bugs are comparably severe.

9.3.4 Encoding Bugs

The encoding bugs for input coding, output coding, state codes and state-symbol product formation can exist not only in an explicit finite state machine but also when the finite state machine is implicit.

Make sure not to use the programmer's state numbers and/or input codes. The behavior of a finite-state machine is invariant under all encodings. Let us assume that the states are numbered from 1 to n. If the states are numbered using an arbitrary permutation, the finite state machine remains unchanged – similarly for input and output codes. Thus, if we present our version of the finite state machine with a different encoding and if the programmer objects to the renaming or claims that behavior is changed as a result, then there must be some encoding bugs in the machine.

The implementation of the fields as a group of bytes or bits gives you the potential size of the code. If the number of code values is less than this potential, then there is an encoding process going on.

Check Your Progress 2

State whether the following statements are true or false:

1. Finite-state machines are not used in time critical applications.
2. Encoding bugs can occur only in an implicit finite state machine.
3. Output actions must be verified independent of states and transitions.
4. Transitions from a dead state to other states are possible.

9.4 STATE TESTING

9.4.1 Impact of Bugs

One or more of the following symptoms indicate the presence of bugs:

1. Incorrect number of states.
2. Incorrect transition for a given state-input combination.
3. Incorrect output for a given transition.
4. Pairs or sets of states that are accidentally made equivalent.
5. States or set of states that are split to create inequivalent duplicates.
6. States or sets of states that have become dead.
7. States or set of states that have become unreachable.

9.4.2 Principles

The strategy for state testing is similar to that of flow graph testing. It is impractical to go through every path in a state graph as is the case with a flow graph. A path in a state graph is a succession of transitions caused due to a sequence of inputs. Coverage is ensured by passing through each link. Assume a state as an initial state and it must be possible to every possibles state and back to the initial state when starting from the initial state. It is still impractical to do this for these reasons:

1. In the early phase of testing it is not possible to thoroughly cover the graph because of bugs.
2. Later, during maintenance, testing objectives are understood, and only a few states and transitions need to be tested. A complete coverage is waste of time.
3. Verification of a long test sequence is difficult.

The starting point of state testing is:

1. Define a set of covering input sequence that get back to the initial state when starting from the initial state.

2. For each step in each input sequence, define the expected next state, the expected transition, and the expected output code.

A set of tests then comprises of three sets of sequences:

1. Input sequences.
2. Corresponding transitions or next-state names.
3. Output sequences.

9.4.3 Limitations and Extensions

Like the flow-graph model, the state-transition coverage in a state-graph model does not ensure complete testing. It is not essential to consider any sequence longer than the total number of states.

Chou defines a hierarchy of paths and methods to produce covers of tests for combining paths. The simplest one is called the "0-switch", which corresponds to testing each transition individually. The next in the hierarchy is the testing of transition sequences comprising of two transitions called "1 switch". The maximum length switch is an n – 1 switch, where n stands for the number of states. This mechanism shows that, in general, a 0 switch cover can catch output errors but may not detect transition errors. The exact theory to catch specified kinds of state-graph errors using sufficient number of tests is still not developed completely. Thus, we have the following experience:

1. Identify the factors that contribute to the state, calculating the total number of states and comparing this to the designer's notion reveals some bugs.
2. Insisting on a justification for all supposedly dead, unreachable and impossible states and transitions catches few more bugs.
3. Insisting on an explicit specification of the transition and output for every combination of input and state catches many more bugs.
4. A set of input sequences that provide coverage of all nodes and links is a mandatory minimum requirement.
5. In executing state tests, it is must to provide the means to record the state sequence resulting from the input sequence and not only outputs that result from the input sequence.

9.4.4 What to Model

The representation of every program using a state graph is possible as every combination of software and hardware can be modeled using these. The state graph is a behavioral model. It is functional rather than structural and is far removed from the code. As a testing method, it is a bottom line method that ignores structural detail to focus on behavior. State testing guarantees bigger payoffs during the test design rather than running, as compared to any other test method. As the tests can be constructed from a design specification in advance of the coding, they help catching deep bugs early which prove to be more expensive during later stages. State testing can prove useful in the following cases:

1. In the processing where the output depends upon the occurrence of one or more sequence of events e.g. detection of specified input sequences, sequential format validation and other situation in which the order of input is important.
2. Most protocols between systems, between humans and machines, between components of a system.

3. Device drivers such as tapes and discs that have complicated retry and recovery procedures if the action depends on state.
4. Transaction flows where the transactions are such that they can stay in the system indefinitely.
5. High-level control functions within an operating system.
6. The behavior of the system w.r.t. resource management and what it will do when various levels of resource utilization are reached.
7. A set of menus and ways that can go from one to other.
8. Whenever a feature is directly explicitly implemented as one or more state-transition tables.

9.4.5 Getting the Data

The majority job of an independent tester's life is to get the data on which the model is to be based. State testing has an intensive data-gathering phase and needs time to resolve issues. This happens because most of the participants don't realize that there's an important state-machine behavior.

9.5 LET US SUM UP

State testing is mainly a functional testing tool whose payoff is best in the early phases of design.

A program cannot have ambiguous transitions or outputs, but a specification can and does. Use a state table to verify the specification's validity. Count the states.

Insist on a specification of transition and output for every combination of input and states. Apply a minimum set of covering tests.

Instrument the transitions to capture the sequence of states and not just the sequence of outputs.

9.6 LESSON END ACTIVITIES

1. Describe transition bugs giving reference to the unreachable and dead states.
2. List the scenarios in which state testing prove to be more useful.
3. What are the limitations of state graph testing? How can they be dealt with?

9.7 KEYWORDS

Dead state: A dead state (or a set of dead states) is a state that once entered cannot be left.

Unreachable state: An unreachable state is the one that cannot be reached. It is not impossible.

Equivalent states: Two states are said to be equivalent if every sequence of inputs starting from one state produces exactly the same sequence of outputs when started from the other state.

Number of states: The number of states in a state graph is the number of states we choose to recognize or model.

State table: A table that specifies the states, inputs, transitions and outputs is called a state table or a state-transition table.

State: State is a combination of circumstances or attributes belonging for the time being to a person or thing.

9.8 QUESTIONS FOR DISCUSSION

1. What are equivalent states? How can they be identified?
2. List down the ways to categorize a state-graph as good or bad.
3. What is a state? Explain the transition of states in a state graph.

Check Your Progress: Model Answers

CYP 1

1. Input alphabet
2. next state (the transition), output (if any)

CYP 2

1. False
2. False
3. True
4. True

9.9 SUGGESTED READINGS

Boris Beizer, *Software Testing Techniques*, Second Edition, Dreamtech Press, 2003.

Myers and Glenford, J., *The Art of Software Testing*, John-Wiley & Sons, 1979.

Roger, S. Pressman, *Software Engineering – A Practitioner's Approach*, 5th Edition, McGraw Hill, 2001.

Marnie, L. Hutcheson, *Software Testing Fundamentals*, Wiley-India, 2007.

UNIT V

LESSON

10

TESTING SPECIALIZED ENVIRONMENTS, ARCHITECTURE AND APPLICATIONS

CONTENTS

10.0 Aims and Objectives

10.1 Introduction

10.2 Testing GUIs

- 10.2.1 GUI Test Strategies
- 10.2.2 Types of GUI Tests
- 10.2.3 Improving GUI Testability

10.3 Testing of Client-Server Architecture

- 10.3.1 Client-Server Software
- 10.3.2 Client-Server Testing Techniques
- 10.3.3 Testing Aspects
- 10.3.4 Measures of Completeness

10.4 Testing Documentation and Help Facilities

- 10.4.1 Software Reviews
- 10.4.2 Test Deliverables
- 10.4.3 Samples of Test Deliverables

10.5 Testing of Real-Time Systems

10.6 Let us Sum up

10.7 Lesson End Activities

10.8 Keywords

10.9 Questions for Discussion

10.10 Suggested Readings

10.0 AIMS AND OBJECTIVES

After studying this lesson, you would be able to understand:

- The Graphical User Interfaces (GUI)
- To test a client-server based architecture
- How to test documentation and help facilities
- How to test real time systems

10.1 INTRODUCTION

The testing methods discussed in preceding sections can be applied across all environments, architectures and applications, but unique guidelines and approaches to testing are sometimes warranted. In this lesson we will discuss testing guidelines for specialized environments, architectures and applications that are commonly by software engineers.

10.2 TESTING GUIs

Graphical User Interfaces (GUIs) present interesting challenges for software engineers. Because of the reusable components provided as a part of GUI development environments, the creation of the user interface has become less time consuming and more precise. But, at the same time, the complexity of GUIs has grown, leading to more difficulty in the design and execution of test cases.

A series of tests can be derived and applied to test the GUIs because many GUIs have the same look and feel. Tests addressing specific data and program objects relevant to the GUI can be derived with the help of a finite state modeling graphs.

Because of a large number of permutations associated with the GUI operations, testing should be done using automated tools. A wide variety of GUI testing tools has been made available in the market over the past few years.

Once a user interface prototype has been created, it must be evaluated to check if it meets the user needs. Evaluation can include a range of tests ranging from informal test drive, in which a user provides impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a group of end-users. Design modifications are made based on user inputs, and the next level of prototype is created. The evaluation cycle continues until no further modifications are required to the interface design.

Although the prototyping technique is effective, but is not always possible to evaluate the quality of a user interface before a prototype is built. If potential problems are uncovered and corrected early, the number of loops through the evaluation cycle will be reduced and development time will shorten. A number of evaluation criteria can be applied to a design model at the time of early design reviews once the design model of the interface has been created. These are:

1. The length and complexity of the written specification of the system and its interface provide an indication of the amount of learning required by users of the system.
2. The number of tasks specified and the average number of actions per task provide an indication of interaction time and the overall efficiency of the system.
3. The number of actions, tasks and systems states indicated by the design model implies the memory load on users of the system.
4. Interface style, help facilities and error handling protocol provide general indication of the complexity of the interface and the degree to which it will be accepted by the user.

Once the prototype is built the designer can collect a variety of qualitative and quantitative data that will assist him in evaluating the interface.

10.2.1 GUI Test Strategies

Test Principles Applied to GUIs

Our proposed approach to testing GUIs is guided by several principles, most of which should be familiar. By following these principles we will develop a test process which is generally applicable for testing any GUI application. Note that the proposed test approach does not cover white-box testing of application code in any depth. This approach concentrates on GUI errors and using the GUI to exercise tests so is very-oriented toward black-box testing.

- ***Focus on errors to reduce the scope of tests:*** We intend to categorise errors into types and design test to detect each type of error in turn. In this way, we can focus the testing and eliminate duplication.
- ***Separation of concerns (divide and conquer):*** By focusing on particular types of error and designing test cases to detect those errors, we can break up the complex problem into a number of simpler ones.
- ***Test design techniques where appropriate:*** Traditional black box test techniques that we would use to test forms based applications are still appropriate.
- ***Layered and staged tests:*** We will organise the test types into a series of test stages. The principle here is that we bring tests of the lowest level of detail in components up front. We implement integration tests of components and test the integrated application last. In this way, we can build the testing up in trusted layers.
- ***Test automation...wherever possible:*** Automation most often fails because of over-ambition. By splitting the test process into stages, we can seek and find opportunities to make use of automation where appropriate, rather than trying to use automation everywhere.

High Level Test Process

An outline test process is presented in Figure 10.1. The high-level test process. We can split the process into three overall phases: Test Design, Test Preparation and Test Execution. In this paper, we are going to concentrate on the first stage: Test Design, and then look for opportunities for making effective use of automated tools to execute tests.

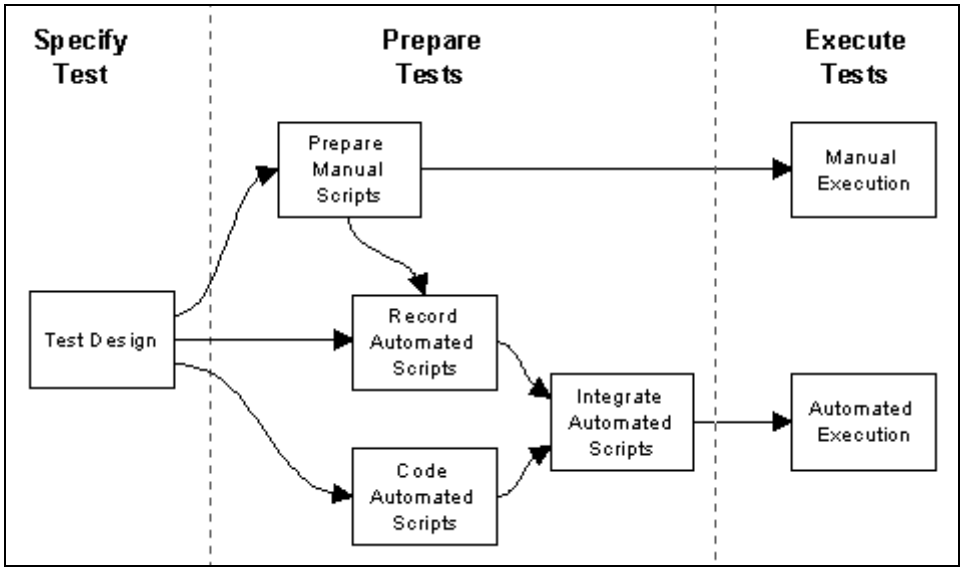


Figure 10.1: The High Level Test Process

Types of GUI errors

We can list some of the multifarious errors that can occur in a client/server-based application that we might reasonably expect to be able to test for using the GUI. The list in Table 10.1 is certainly not complete, but it does demonstrate the wide variety error types. Many of these errors relate to the GUI, others relate to the underlying functionality or interfaces between the GUI application and other client/server components.

Table 10.1: The Variety of Errors Found in GUI Applications

• Data validation • Incorrect field defaults • Mishandling of server process failures • Mandatory fields, not mandatory • Wrong fields retrieved by queries • Incorrect search criteria • Field order • Multiple database rows returned, single row expected • Currency of data on screens • Window object/DB field correspondence	• Correct window modality? • Window system commands not available/don't work • Control state alignment with state of data in window? • Focus on objects needing it? • Menu options align with state of data or application mode? • Action of menu commands aligns with state of data in window • Synchronisation of window object content • State of controls aligns with state of data in window?

By targeting different categories of errors in this list, we can derive a set of different test types that focus on a single error category of errors each and provide coverage across all error types.

The four stages are summarised in Table 10.2 below. We can map the four test stages to traditional test stages as follows:

- ***Low level*** - maps to a unit test stage.
- ***Application*** - maps to either a unit test or functional system test stage.
- ***Integration*** - maps to a functional system test stage.
- ***Non-functional*** - maps to non-functional system test stage.

The mappings described above are approximate. Clearly there are occasions when some 'GUI integration testing' can be performed as part of a unit test. The test types in 'GUI application testing' are equally suitable in unit or system testing. In applying the proposed GUI test types, the objective of each test stage, the capabilities of developers and testers, the availability of test environment and tools all need to be taken into consideration before deciding whether and where each GUI test type is implemented in your test process.

The GUI test types alone do not constitute a complete set of tests to be applied to a system. We have not included any code-based or structural testing, nor have we considered the need to conduct other integration tests or non-functional tests of performance, reliability and so on. Your test strategy should address all these issues.

Table 10.2: Proposed GUI Test Stages

Stage	Test Types
Low Level	• Checklist testing • Navigation
Application	• Equivalence Partitioning • Boundary Values • Decision Tables • State Transition Testing
Integration	• Desktop Integration • C/S Communications • Synchronisation
Non-Functional	• Soak testing • Compatibility testing • Platform/environment

10.2.2 Types of GUI Tests

Checklist Testing

Checklists are a straightforward way of documenting simple re-usable tests. Not enough use is made of checklist driven testing, perhaps because it is perceived to be trivial, but it is their simplicity that makes them so effective and easy to implement. They are best used to document simple checks that can be made on low-level components. Ideally, these checks can be made visually, or by executing very simple or easily remembered series of commands. The types of checks that are best documented in this way are:

- Programming/GUI standards covering standard features such as:
  - ❖ window size, positioning, type (modal/non-modal)
  - ❖ standard system commands/buttons (close, minimise, maximise etc.)
- Application standards or conventions such as:
  - ❖ standard OK, cancel, continue buttons, appearance, colour, size, location
  - ❖ consistent use of buttons or controls
  - ❖ object/field labelling to use standard/consistent text.

These checks should be both simple to document and execute and can be used for standalone components so that programmers may make these checks before they release the code for integration.

Navigation Testing

In the context of a GUI, we can view navigation tests as a form of integration testing. Typically, programmers create and test new windows in isolation. Integration of a new window into an application requires that the application menu definition and invocations of the window from other windows be correctly implemented. The build strategy determines what navigation testing can be done and how. To conduct meaningful navigation tests the following are required to be in place:

- An application backbone with at least the required menu options and call mechanisms to call the window under test.

- Windows that can invoke the window under test.
- Windows that are called by the window under test.

Obviously, if any of the above components are not available, stubs and/or drivers will be necessary to implement navigation tests. If we assume all required components are available, what tests should we implement? We can split the task into steps:

- For every window, identify all the legitimate calls to the window that the application should allow and create test cases for each call.
- Identify all the legitimate calls from the window to other features that the application should allow and create test cases for each call.
- Identify reversible calls, i.e. where closing a called window should return to the 'calling' window and create a test case for each.
- Identify irreversible calls i.e. where the calling window closes before the called window appears.

There may be multiple ways of executing a call to another window i.e. menus, buttons, keyboard commands. In this circumstance, consider creating one test case for each valid path by each available means of navigation.

Note that navigation tests reflect only a part of the full integration testing that should be undertaken. These tests constitute the 'visible' integration testing of the GUI components that a 'black box' tester should undertake.

Application Testing

Application testing is the testing that would normally be undertaken on a forms-based application. This testing focuses very much on the behaviour of the objects within windows. The approach to testing a window is virtually the same as would be adopted when testing a single form. The traditional black-box test design techniques are directly applicable in this context.

A brief summary of the most common techniques and some guidelines for their use with GUI windows are presented in the table below:

Table 10.3: Traditional Test Techniques

Technique	Used to test
Equivalence Partitions and Boundary Value Analysis	• Input validation • Simple rule-based processing
Decision Tables	• Complex logic or rule-based processing
State-transition testing	• Applications with modes or states where processing behaviour is affected • Windows where there are dependencies between objects in the window.

Desktop Integration Testing

It is rare for a desktop PC or workstation to run a single application. Usually, the same machine must run other bespoke applications or shrink wrapped products such as a word processor, spreadsheet, electronic mail or Internet based applications. Client/server systems assume 'component based' architecture so they often treat other products on the desktop as components and make use of features of these products by calling them as components directly or through specialist middleware.

We define desktop integration as the integration and testing of a client application with these other components. Because these interfaces may be hidden or appear 'seamless' when working, the tester usually needs to understand a little more about the technical implementation of the interface before tests can be specified. The tester needs to know what interfaces exist, what mechanisms are used by these interfaces and how the interface can be exercised by using the application user interface.

To derive a list of test cases the tester needs to ask a series of questions for each known interface:

- Is there a dialogue between the application and interfacing product (i.e. a sequence of stages with different message types to test individually) or is it a direct call made once only?
- Is information passed in both directions across the interface?
- Is the call to the interfacing product context sensitive?
- Are there different message types? If so, how can these be varied?

In principle, the tester should prepare test cases to exercise each message type in circumstances where data is passed in both directions. Typically, once the nature of the interface is known, equivalence partitioning, boundary values analysis and other techniques can be used to expand the list of test cases.

Client/Server Communication Testing

Client/Server communication testing complements the desktop integration testing. This aspect covers the integration of a desktop application with the server-based processes it must communicate with. The discussion of the types of test cases for this testing is similar to section Desktop Integration, except there should be some attention paid to testing for failure of server-based processes.

In the most common situation, clients communicate directly with database servers. Here the particular tests to be applied should cover the various types of responses a database server can make. For example:

- Logging into the network, servers and server-based DBMS.
- Single and multiple responses to queries.
- Correct handling of errors (where the SQL syntax is incorrect, or the database server or network has failed)
- Null and high volume responses (where no rows or a large number of rows are returned).

The response times of transactions that involve client/server communication may be of interest. These tests might be automated, or timed using a stopwatch, to obtain indicative measures of speed.

Synchronisation Testing

There may be circumstances in the application being tested where there are dependencies between different features. One scenario is when two windows are displayed; a change is made to a piece of data on one window and the other window needs to change to reflect the altered state of data in the database. To accommodate such dependencies, there is a need for the dependent parts of the application to be synchronised.

Examples of synchronisation are when:

- The application has different modes - if a particular window is open, then certain menu options become available (or unavailable).
- If the data in the database changes and these changes are notified to the application by an unsolicited event to update displayed windows.
- If data on a visible window is changed and makes data on another displayed window inconsistent.

In some circumstances, there may be reciprocity between windows. For example, changes on window A trigger changes in window B and the reverse effect also applies (changes in window B trigger changes on window A).

In the case of displayed data, there may be other windows that display the same or similar data which either cannot be displayed simultaneously, or should not change for some reason. These situations should be considered also. To derive synchronisation test cases:

- Prepare one test case for every window object affected by a change or unsolicited event and one test case for reciprocal situations.
- Prepare one test case for every window object that must not be affected - but might be.

Non-functional Testing

The tests described in the previous sections are functional tests. These tests are adequate for demonstrating the software meets it's requirements and does not fail. However, GUI applications have non-functional modes of failure also. We propose three additional GUI test types (that are likely to be automated).

Soak Testing

In production, systems might be operated continuously for many hours. Applications may be comprehensively tested over a period of weeks or months but are not usually operated for extended periods in this way. It is common for client application code and bespoke middleware to have memory-leaks. Soak tests exercise system transactions continuously for an extended period in order to flush out such problems.

These tests are normally conducted using an automated tool. Selected transactions are repeatedly executed and machine resources on the client (or the server) monitored to identify resources that are being allocated but not returned by the application code.

Compatibility Testing

Whether applications interface directly with other desktop products or simply co-exist on the same desktop, they share the same resources on the client. Compatibility Tests are (usually) automated tests that aim to demonstrate that resources that are shared with other desktop products are not locked unnecessarily causing the system under test or the other products to fail.

These tests normally execute a selected set of transactions in the system under test and then switch to exercising other desktop products in turn and doing this repeatedly over an extended period.

Platform/Environment Testing

In some environments, the platform upon which the developed GUI application is deployed may not be under the control of the developers. PC end-users may have a variety of hardware types such as 486 and Pentium machines, various video drivers,

Microsoft Windows 3.1, 95 and NT. Most users have PCs at home now-a-days and know-how to customise their PC configuration. Although your application may be designed to operate on a variety of platforms, you may have to execute tests of these various configurations to ensure when the software is implemented; it continues to function as designed. In this circumstance, the testing requirement is for a repeatable regression test to be executed on a variety of platforms and configurations. Again, the requirement for automated support is clear so we would normally use a tool to execute these tests on each of the platforms and configurations as required.

10.2.3 Improving GUI Testability

The GUI Testing Challenge

It is clear that GUIs present a challenge to testers because they appear to be inherently more difficult to test. The flexibility of GUIs invites programmers to pass on this flexibility to end users in their applications. Consequently, users can exercise the application code in ways never envisaged by the programmers and which are likely to be released untested.

If testability is the ease with which a tester can specify, prepare, execute and analyze tests, it is arguable that it is possible for programmers to build systems using GUIs that cannot be tested.

It is difficult to specify tests because much of the underlying functionality in a GUI application is undocumented. Because of the event-driven nature of GUIs, a considerable amount of programming effort is expended on dealing with hidden interactions that come to light during the informal programmer testing so tend to go undocumented.

It is difficult to prepare tests because the number tests required to exercise paths through the application which a user might follow has escalated dramatically. If we consider using menus, function keys and mouse movements to exercise system features, the number of tests increased further.

It is difficult to execute tests. Using a manual pointing device is virtually unrepeatable and certainly error prone. Creating tests which stimulate hidden interactions, set or amend visible (or invisible) GUI objects is troublesome. Separating tests of application code from the GUI elements of the operating system is tricky.

It is difficult to analyse tests because there is constant change on the screen and behind the screen. Windows on which results are displayed may appear and all other visible windows may be refreshed simultaneously making visual inspection difficult. Expected results may not be directly displayed but on hidden windows. Attributes of objects to be verified may be invisible or difficult to detect by eye. Windows that display invalid results may be hidden by other windows or on windows that are minimised.

GUI Design for Testability

We make the following recommendations to GUI designers aimed at improving testability. We suggest that the most straightforward way of implementing them is to include checks on these design issues in checklist test cases. Some of these recommendations impact the freedom users have to use software in certain ways, but we believe that if the application structure and organisation is well designed, the user will have little need to make unusual choices.

1. Where applications have modes of operation so that some features become meaningless or redundant, then these options on menus should be greyed-out or disabled in some other way.

2. Unless there are specific requirements to display the same data on multiple windows the designer should avoid having to build in dependencies between windows to eliminate 'displayed data' inconsistencies.
3. Navigation between windows should be hierarchic, (in preference to anarchic) to minimise the number of windows that might be open at once and to reduce the number of paths through the system.
4. Unless there is an impact on usability, windows should be modal to reduce the number of paths through the system and reduce window testing to a simpler, forms-like test process.
5. Unless there is an impact on usability, dependencies between objects on windows should be avoided or circumvented by splitting user transactions into multiple modal windows.
6. The number of system commands (maximise, minimise, close, restore) available on windows should be reduced to a minimum.
7. Functionality which is accessed by equivalent button clicks, function keys and menu options should be implemented using the same function-call to reduce the possibility of errors and the need to always test all three mechanisms.
8. Instrumentation should be implemented in code to provides information on application interfaces to other desktop products or server-based processes and should be an option which can be turned on or off by testers.
9. Instrumentation should be implemented to provide information on the content of unsolicited events from other applications and also to simulate these unsolicited events for test purposes.

10.3 TESTING OF CLIENT-SERVER ARCHITECTURE

Client-server architectures represent a significant challenge for software tests. The distributed nature of client/server environments, the performance issue associated with transaction processing, the potential presence of a number of different hardware platforms, the complexity of network communication, the need to service multiple clients from a centralized (or distributed) database, and the coordination requirements imposed on the server all combine to make testing of client/server software architecture considerably more difficult than standalone applications. In fact, recent studies indicate a significant increase in the testing time and cost when client/server environments are developed.

In general, the testing of client/server software occurs at three different levels: (i) individual client applications are tested in a disconnected mode, the operation of the server and the underlying network are not considered; (ii) the client software and associated server applications are tested in concert, but network operations are not explicitly exercised; (iii) the complete client/server architecture including the network operations and performance, is tested.

Although many different types of tests are conducted at each of these levels of detail, the following testing approaches are commonly used:

- ***Application function tests:*** The functionality of client applications is tested in a standalone fashion.
- ***Server tests:*** The coordinated and data management functions of the server are tested. Server performance (overall response time and data throughput) is also considered.

- ***Database tests:*** The accuracy and integrity of data stored by the server is tested. Transactions posted by client applications are examined to ensure that data are properly stored, updated and retrieved. Archiving is also tested.
- ***Transaction tests:*** A series of tests are created to ensure that each class of transactions is processed according to requirements. Tests focus on the correctness of processing and also on performance issues (e.g. transaction processing times and transaction volumes).
- ***Network communication tests:*** These tests verify that communication among the nodes of the network occurs correctly and that message passing, transactions, and related network traffic occur without error. Network security tests may also be conducted as part of these tests.

In order to accomplish these testing approaches, it is recommended to develop operation profiles derived from client/server usage scenarios. An operational profile indicates how different types of users interoperate with the client/server system i.e. the profiles provide a “pattern of usage” that can be applied when tests are designed and executed.

10.3.1 Client Server Software

Client server software requires specific forms of testing to prevent or predict catastrophic errors. Servers go down, records lock, I/O (Input/Output) errors and lost messages can really cut into the benefits of adopting this network technology. Testing addresses system performance and scalability by understanding how systems respond to increased workloads and what causes them to fail.

Software testing is more than just review. It involves the dynamic analysis of the software being tested. It instructs the software to perform tasks and functions in a virtual environment. This examines compatibility, capability, efficiency, reliability, maintainability, and portability. A certain amount of faults will probably exist in any software. However, faults do not necessarily equal failures. Rather, they are areas of slight unpredictability that will not cause significant damage or shutdown. They are more errors of semantics. Therefore, testing usually occurs until a company reaches an acceptable defect rate that doesn’t affect the running of the program or at least won’t until an updated version has been tested to correct the defects.

Since client-server technology relies so heavily on application software and networking, testing is an important part of technology and product development. There are two distinct approaches when creating software tests. There is black box testing and white or glass box testing. Black box testing is also referred to as functional testing. It focuses on testing the internal machinations of whatever is being tested, in our case, a client or server program. When testing software, for example, black box tests focus on I/O. The testers know the input and predicted output, but they do not know how the program arrives at its conclusions. Code is not examined, only specifications are.

Black box testing does not require special knowledge of specific languages from the tester. The tests are unbiased because designers and testers are independent of each other. They are primarily conducted from the user perspective to ensure usability. However, there are also some disadvantages to black box testing. The tests are difficult to design and can be redundant. Also, many program paths go uncovered since it is realistically impossible to test all input streams. It would simply take too long.

White box testing is also sometimes referred to as glass box testing. It is a form of structural testing that is also called clear box testing or open box testing. As expected,

it is the opposite of black box testing. It focuses on the internal workings of the program and uses programming code to examine outputs. Furthermore, the tester must know what the program is supposed to do and how it's supposed to do it. Then, the tester can see if the program strays from its proposed goal. For software testing to be complete both functional/black and structural/white/glass box testing must be conducted.

10.3.2 Client Server Testing Techniques

Risk Driven Testing and Performance Testing

There are a variety of testing techniques that are particularly useful when testing client and server programs. Risk driven testing is time sensitive, which is important in finding the most important bugs early on. It also helps because testing is never allocated enough time or resources. Companies want to get their products out as soon as possible. The prioritization of risks or potential errors is the engine behind risk driven testing. In risk driven testing the tester takes the system parts he/she wants to test, modules or functions, for example, and examines the categories of error impact and likelihood. Impact, the first category, examines what would happen in the event of a break down. For example, would entire databases be wiped out or would the formatting just be a little off? Likelihood estimates the probability of this failure in the element being tested. Risk driven testing prioritizes the most catastrophic potential errors in the service of time efficiency.

Performance testing is another strategy for testing client and server programs. Simply put, performance testing evaluates system components, such as software, around specific performance parameters, such as resource utilization, response time, and transaction rates. It is also called load testing or stress testing. In order to performance test a client-server application, several key pieces of information must be known.

For example, the average number of users working simultaneously on a system must be quantified, since performance testing most commonly tests performance under workload stress. Testers should also determine maximum or peak user performance or how the system operates under maximum workloads. Bandwidth is another necessary bit of information, as is most users' most frequent actions. Performance testing also validates and verifies other performance parameters such as reliability and scalability. Performance testing can establish that a product lives up to performance standards necessary for commercial release. It can compare two systems to determine which one performs better. Or they can use profilers to determine the program's behavior as it runs. This determines which parts of the program might cause the most trouble and it establishes thresholds of acceptable response times.

10.3.3 Testing Aspects

Unit testing, Integration testing, and System testing

There are different types of software testing that focus on different aspects of IT architecture. Three in particular are particularly relevant to client server applications. These are unit testing, integration testing, and system testing. A unit is the smallest testable component of a program. In object–oriented programming, which is increasingly influencing client-server applications, the smallest unit is a class. Modules are made up of units.

Unit testing isolates small sections of a program (units) and tests the individual parts to prove they work correctly. They make strict demands on the piece of code they are testing. Unit testing documentation provides records of test cases that are designed to incorporate the characteristics that will make the unit successful. This documentation

also contains positive and negative uses for the unit as well as what negative behaviors the unit will trap. However, unit testing won't catch all errors. It must be used with other testing techniques. It is only a phase of three-layer testing, of which unit testing is the first.

Integration testing, sometimes called I&T (Integration and Testing), combines individual modules and tests them as a group. These test cases take modules that have been unit tested, they test this input with a test plan. The output is the integrated system, which is then ready for the final layer of testing, system testing. The purpose of integration testing is to verify functionality, performance, and reliability. There are different types of integration testing models. For example, the Big Bang model is a time saver by combining unit-tested modules to form an entire software program (or a significant part of one). This is the 'design entity' that will be tested for integration.

However, record of test case results is of the essence, otherwise further testing will be very complicated. Bottom up integrated testing tests all the low, user level modules, functions and procedures. Once these have been integrated and tested, the next level of modules can be integrated an tested. All modules at each level must be operating at the same level for this type of testing to be worthwhile. In object-oriented programming, of which client server applications increasingly are, classes are encapsulations of data attributes and functions. Classes require the integration of methods. Ultimately, integration testing reveals any inconsistencies within or between assemblages or the groupings of modules that are integrated through testing plans and outputs.

System testing is the final layer of software testing. It is conducted once the system has been integrated. Like integration testing, it falls within the category of black box testing. Its input is the integrated software elements that have passed integration testing and the integration of the software system with any hardware systems it may apply to. System testing detects inconsistencies between assemblages (thereby testing integration) and in the system as its own entity. System testing is the final testing front and therefore the most aggressive. It runs the system to the point of failure and is characterized as destructive testing. Here are some of the areas that system testing covers usability, reliability, maintenance, recover, compatibility, and performance.

10.3.4 Measures of Completeness

In software testing there are two measures of completeness, code coverage and path coverage. Code coverage is a white box testing technique to determine how much of a program's source code has been tested. There are several fronts on which code coverage is measured.

For example, statement coverage determines whether each line of code has been executed and tested. Condition coverage checks the same for each evaluation point. Path coverage establishes whether every potential route through a segment of code has been executed and tested. Entry/Exit coverage executes and tests every possible call to a function and return of a response. Code coverage provides a final layer of testing because it searches for the errors that were missed by the other test cases. It determines what areas have not been executed and tested and creates new test cases to analyze these areas. In addition, it identifies redundant test cases that won't increase coverage.

Check Your Progress 1

Fill in the blanks:

1. Tests addressing specific data and program objects relevant to the GUI can be derived with the help of .....
2. Network security tests may also be conducted as part of ..... tests.

10.4 TESTING DOCUMENTATION AND HELP FACILITIES

The term software testing conjures images of large number of test cases prepared to exercise computer programs and the data that they manipulate. The testing must also extend to the third element of the software configuration – documentation.

Errors in documentation can be devastating to the acceptance of the program as errors in data or source code. It becomes very frustrating if one follows a user-guide or an online help facility exactly and ultimately gets results or behaviors that do not match with those predicted in the documentation. Thus, it is necessary to test the documentation and it must be a part of every software test plan.

Documentation testing can be approached in two phases: (1) review and inspection, examines the documentation for editorial clarity and (2) live test, uses the documentation in conjunction with the use of the actual program.

10.4.1 Software Reviews

Software reviews are filers that are applied through the stages of the software engineering process to report errors and defects that can be removed at that stage. It further refines the activities of analysis, design and coding.

Various types of reviews are conducted as a part of software engineering. Each of these has its own importance. An informal meeting at coffee is a kind of review to discuss technical issues. A formal presentation of software design to customers, management and technical staff is also a kind of review. A formal technical review or walkthrough is the most effective of all these reviews from the view of QA. It is conducted by software engineers, for software engineers to improve the software quality.

10.4.2 Test Deliverables

Test Documentation: Documentation describing plans for, or results of, the testing of a system or component, Types include test case specification, test incident report, test log, test plan, test procedure, test report.

Software Testing documentation, or Test Deliverables, may consist of the following documents:

- ***Master test plan:*** Sometimes it is possible to write separate documents for test planning: Unit test plan, Integration test plan, System test plan and Acceptance test plan.
- ***Test plan:*** A high-level document that defines a software testing project so that it can be properly measured and controlled. It defines the test strategy and organized elements of the test life cycle, including resource requirements, project schedule, and test requirements.
- ***Test cases design:*** A set of test inputs, executions, and expected results developed for a particular objective.

- ***Test procedures:*** A document, providing detailed instructions for the [manual] execution of one or more test cases. Often called - a manual test script.
- ***Test logs:*** A chronological record of all relevant details about the execution of a test.
- ***Test data:*** The actual (set of) values used in the test or that are necessary to execute the test.
- ***Test summary report***
- ***Automated test scripts***
- ***Incident reports***
- ***Incident log***

But as a minimum you must have test strategy, test cases and test summary report.

10.4.3 Samples of Test Deliverables

A sample of a Master Software Test Plan document contents

1. Introduction
   - 1.1 Purpose
   - 1.2 Background
   - 1.3 Scope
   - 1.4 Project Identification
2. Software Structure
   - 2.1 Software Risk Issues
3. Test Requirements
   - 3.1 Features Not to Test
   - 3.2 Metrics
4. Test Strategy
   - 4.1 Test Cycles
   - 4.2 Planning Risks and Contingencies
5. Testing Types
   - 5.1 Functional Testing
   - 5.2 User Interface Testing
   - 5.3 Configuration Testing
   - 5.4 Installation Testing
   - 5.5 Volume Testing
   - 5.6 Performance Testing
   - 5.7 Tools
6. Resources
   - 6.1 Staffing
   - 6.2 Training Needs

Contd….

7. Project Milestones
8. Deliverables
   - 8.1 Test Assets
   - 8.2 Exit criteria
   - 8.3. Test Logs and Defect Reporting
9. References

A good test strategy is the most important and can in some cases replace all test plan documents. The purpose of a test strategy is to clarify the major tasks and challenges of the test project.

Sample of Test Strategy document contents

1. INTRODUCTION
   - 1.1 PURPOSE
   - 1.2 FUNCTIONAL OVERVIEW
   - 1.3 CRITICAL SUCCESS FACTOR
   - 1.4 TESTING SCOPE (TBD)
     - Inclusions
     - Exclusions
   - 1.5 TEST COMPLETION CRITERIA
2. TIMEFRAME
3. RESOURCES
   - 3.1 TESTING TEAM SETUP
   - 3.2 HARDWARE REQUIREMENTS
   - 3.3 SOFTWARE REQUIREMENTS
4. APPLICATION TESTING RISKS PROFILE
5. TEST APPROACH
   - 5.1 STRATEGIES
   - 5.2 GENERAL TEST OBJECTIVES
   - 5.3 APPLICATION FUNCTIONALITY
   - 5.4 APPLICATION INTERFACES
   - 5.5 TESTING TYPES
     - 5.5.1 Stability
     - 5.5.2 System
     - 5.5.3 Regression
     - 5.5.4 Installation
     - 5.5.5 Recovery

Contd…..

5.5.6 Configuration
5.5.7 Security
6. BUSINESS ARES FOR SYSTEM TEST
7. TEST PREPARATION
7.1 TEST CASE DEVELOPMENT
7.2 TEST DATA SETUP
7.3 TEST ENVIRONMENT
7.3.1 Database Restoration Strategies
8. TEST EXECUTION
8.1 TEST EXECUTION PLANNING
8.2 TEST EXECUTION DOCUMENTATION
8.3 PROBLEM REPORTING
9. STATUS REPORTING
9.1 TEST EXECUTION PROCESS
9.2 PROBLEM STATUS
10. HANDOVER FOR USER ACCEPTANCE TEST TEAM
11. DELIVERABLES
12. APPROVALS
13. APPENDICES
13.1 APPENDIX A (BUSINESS PROCESS RISK ASSESSMENT)
13.2 APPENDIX B (TEST DATA SETUP)
13.3 APPENDIX C (TEST CASE TEMPLATE)
13.4 APPENDIX D (PROBLEM TRACKING PROCESS)

Sample of Test Evaluation Report document contents

1. Objectives
2. Scope
3. References
4. Introduction
5. Test Coverage
6. Code Coverage
7. Suggested Actions
8. Diagrams

User Acceptance Test (UAT) Plan Table of Contentss

1. INTRODUCTION
    1.1 PURPOSE
    1.2 FUNCTIONAL OVERVIEW
    1.3 CRITICAL SUCCESS FACTORS
    1.4 UAT SCOPE
    1.5 TEST COMPLETION CRITERIA
2. TIMEFRAME
3. RESOURCES
    3.1 TESTING TEAM
    3.2 HARDWARE TESTING REQUIREMENTS
    3.3 SOFTWARE TESTING REQUIREMENTS
4. TEST APPROACH
    4.1 TEST STRATEGY
    4.2 GENERAL TEST OBJECTIVES:
    4.3 BUSINESS AREAS FOR SYSTEM TEST
    4.4 APPLICATION INTERFACES
5. TEST PREPARATION
    5.1 TEST CASE DEVELOPMENT
    5.2 TEST DATA SETUP
    5.3 TEST ENVIRONMENT
6. UAT EXECUTION
    6.1 PLANNING UAT EXECUTION
    6.2 TEST EXECUTION DOCUMENTATION
    6.3 ISSUE REPORTING
7. HANDOVER FOR UAT ACCEPTANCE COMMITTEE
8. ACCEPTANCE COMMITTEE
9. DELIVERABLES
10. APPROVALS
11. APPENDICES
    11.1 APPENDIX A (TEST CASE TEMPLATE)
    11.2 APPENDIX B (SEVERITY STRATEGY)
    11.3 APPENDIX C (ISSUE LOG)

Check Your Progress 2

State whether the following statements are true or false:

1. It is possible to test the client/server architecture only in the standalone environment and not in the distributed one.
2. Errors in documentation can lead to errors in data or source code.
3. Test cases related to documentation testing can be included in every software test plan.
4. Software reviews are conducted at different staged to get rid of errors encountered at that stage of software evolution.

10.5 TESTING OF REAL-TIME SYSTEMS

The time-dependent, asynchronous nature of many real-time applications adds a new potential difficult element to the testing mix-time. Test case designer must consider both the conventional test cases and also event handling, timing of the data and the parallelism of the tasks that handle the data. In many situations, a real time system in one state will generate correct results when fed with test data, while the same system may lead to error using the same test data while in another state.

Also, the relation between the real-time software and its hardware components can also cause testing problems. Software tests must consider the impact of hardware faults on software processing. Such faults can be extremely difficult to simulate realistically.

Comprehensive test case design methods for real-time systems continue to evolve. However, a four-step strategy can be proposed:

- ***Task Testing:*** The first step in testing real-time software is to test each task independently i.e. conventional test cases are designed and executed for each task. Each task is executed independently during these tests. This type of testing uncovers logical and functional errors but not the ones related to timing and behavior.
- ***Behavioral Testing:*** Using system models created with automated tools, we can simulate the behavior of a real-time system and examine its behavior as a result of external events. These analysis activities can serve as the basis for the test case design which are carried out when the real-time software has been built.
- ***Inter-task Testing:*** Once errors in individual tasks have been isolated from the errors in the system behavior, testing shifts to the time-related errors. Asynchronous tasks are tested with different data rates and processing load to determine if inter-task synchronization errors will occur. In addition, tasks which communicate using message queues or data store are tested to uncover errors in the sizing of these data storage areas.
- ***System Testing:*** Full ranges of tests are carried out after the software and hardware are integrated to reveal the errors at the software/hardware interface. Most real-time system process interrupts. Thus, testing the handling of these Boolean events is essential.

Also, global data areas that are used to transfer information as part of interrupt processing should be tested to assess the potential for the generation of side effects.

10.6 LET US SUM UP

Specialized testing methods encompass a broad array of software capabilities and application areas.

Testing for graphical user interfaces, client/server architectures, documentation and help facilities, and real-time systems each require specialized guidelines and techniques.

10.7 LESSON END ACTIVITIES

1. Why it is difficult to test GUI based applications? How are they tested?
2. What is client-server architecture? What are the problems encountered while testing these applications?
3. List the various testing approaches of client-server architecture based applications.

10.8 KEYWORDS

Informal Meeting: An informal meeting at coffee is a kind of review to discuss technical issues.

Formal Presentation: A formal presentation of software design to customers, management and technical staff is also a kind of review.

Formal Technical Review: A formal technical review or walkthrough is the most effective of all these reviews from the view of QA.

GUI: Graphical User Interface

10.9 QUESTIONS FOR DISCUSSION

1. Discuss the strategy for testing real-time systems.
2. How is the testing of real-time systems different from the other applications?
3. Discuss the different software review techniques.

Check Your Progress: Model Answers

CYP 1

1. Finite state modeling graphs
2. Network communication

CYP 2

1. False
2. True
3. True
4. True

10.10 SUGGESTED READINGS

Boris Beizer, *Software Testing Techniques*, Second Edition, Dreamtech Press, 2003.

Myers and Glenford, J., *The Art of Software Testing*, John-Wiley & Sons, 1979.

Roger, S. Pressman, *Software Engineering – A Practitioner's Approach*, 5th Edition, McGraw Hill, 2001.

Marnie, L. Hutcheson, *Software Testing Fundamentals*, Wiley-India, 2007.

LESSON

11

TESTING TACTICS AND DEBUGGING

CONTENTS

11.0 Aims and Objectives

11.1 Introduction

11.2 Strategic Approach to Testing and Strategic Issues

- 11.2.1 Verification and Validation
- 11.2.2 Organizing for Software Testing
- 11.2.3 Software Testing Strategy
- 11.2.4 Strategic Issues

11.3 Unit Testing

- 11.3.1 Advantages of Unit Testing

11.4 Integration Testing

- 11.4.1 Top-down Integration
- 11.4.2 Bottom-up Integration
- 11.4.3 Regression Testing
- 11.4.4 Smoke Testing
- 11.4.5 Documentation for Integration Testing

11.5 Validation Testing

- 11.5.1 Configuration Review
- 11.5.2 Alpha and Beta Testing

11.6 System Testing

- 11.6.1 Recovery Testing
- 11.6.2 Security Testing
- 11.6.3 Stress Testing
- 11.6.4 Performance Testing

11.7 Debugging

- 11.7.1 Debugging Techniques
- 11.7.2 Debugging Approaches

11.8 Let us Sum up

11.9 Lesson End Activities

11.10 Keywords

11.11 Questions for Discussion

11.12 Suggested Readings

11.0 AIMS AND OBJECTIVES

After studying this lesson, you would be able to understand:

- Strategic approach towards testing and issues
- Unit testing, system testing, validation testing, integration testing and its concepts
- Debugging approaches and processes

11.1 INTRODUCTION

The way in which test cases are executes is as important as the way in which the test cases are designed. A strategy for software testing is developed by the project manager, software engineers and testing specialists.

A proper testing strategy is important because often testing accounts far more project effort than any other software engineering activity. If it is conducted arbitrarily, it results in the wastage of time and effort and can lead to unnoticed errors later on.

Testing normally begins with testing individual small components which are later integrated to carry out the integration testing of the complete software as a whole. Finally, a series of high order test cases are executed once the program is fully operational to uncover the errors in the requirements.

A test specification contains the software team's approach to testing by defining a plan to describe the overall strategy and a procedure that describes specific testing steps and the tests that will be conducted.

11.2 STRATEGIC APPROACH TO TESTING AND STRATEGIC ISSUES

A template for software testing must be designed in advance so that a set of steps, into which we can place specific test case design techniques and testing methods, are defined.

A number of testing strategies have been defined which provide the developer with a template for testing. These strategies have the following characteristics:

- Testing begins at the component level and works outward towards the integration of the entire computer program.
- Different testing techniques can be adopted at different points of time.
- Testing is conducted by the software developer and an independent testing group for large software.
- Testing is different from debugging but debugging must be included in any testing strategy.

A testing strategy must include low-level tests to verify small source code segments as well as high-level tests to validate major system functions against customer requirements. Steps in the defining the test strategy must account for the amount of work completed and the problems to surface as early as possible.

11.2.1 Verification and Validation

Software testing is often referred to as Verification and Validation. Verification refers to the set of activities that ensure that the software correctly implements a specific function. Validation refers to a different set of activities that ensure that the software that has been built is traceable to the customer requirements

Verification: Are we building the product right?

Validation: Are we building the right product?

The V&V comprises of a lot of activities that have already been covered as a part of Software Quality Assurance (SQA).

V&V include activities like Formal Technical Reviews (FTR), quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, development testing, qualification testing and installation testing, etc.

11.2.2 Organizing for Software Testing

Although software analysis and design are constructive activities from a psychological perspective, testing can be considered as a destructive task. Thus, if a software developer is made to test his/her own program, there are chances that he/she might demonstrate that the program is working as per the customer requirement and is error free, rather than uncovering errors.

This may lead erroneous conceptions like: (1) developer should not do any testing at all, (2) software must be tested by third-party testers mercilessly and (3) the testers will be involved only when the testing begins. Each of these interpretations is wrong.

The software developer is responsible for testing individual components of the program ensuring that each of these performs the functions that it is intended/ designed to be doing. In many cases, the developers must also carry out integration testing of the complete program structure. Only after the software architecture is complete, the Independent Test Group (ITG) is involved.

The role of an independent test group is to remove the problems associated with the builder testing the build. Independent testing removes the conflict of interests that may be present otherwise.

The ITG is a part of the software development team i.e. it gets involved at the time of specification and remains involved throughout the project. At times, they may be a part of the SQA group, which leads to higher degree of independence.

11.2.3 Software Testing Strategy

The strategy of software testing can be viewed in the context of a spiral model as shown in Figure 11.1. Unit testing begins at the vortex of the spiral and lays emphasis on each unit of the software as implemented in the source code. Testing progresses by moving outward along the spiral to integration testing, this focuses on design and construction of the software architecture. One step outward lies the validation testing, where requirements captured during the requirements analysis stage are validated against the software built. Finally, we arrive at the system testing, where the software and the other system elements are tested as a whole.

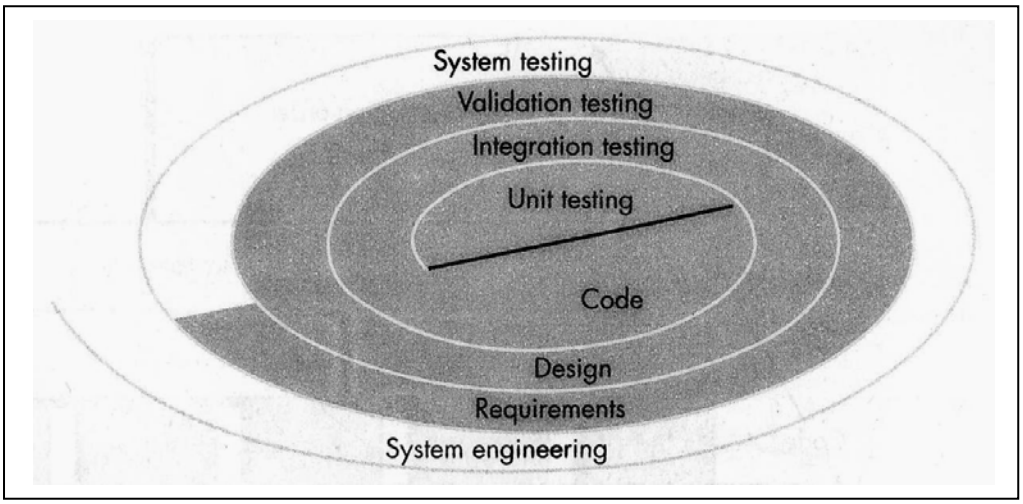


Figure 11.1: Testing Strategy

Testing is carried out in a series of four steps. Initially, the tests focus on individual components, guaranteeing that they function properly as a unit. Hence, the name unit testing. It involves white-box testing to ensure complete coverage and error detection. Next, the components must be assembled to from the complete software. Integration testing addresses the issues related to problem verification and program construction. It mostly includes black-box testing technique. After the software is integrated, validation criteria are tested. Validation testing provides final assurance that software meets all functional, behavioral and performance requirements. It uses black-box testing techniques. The last high-order step is system testing. It verifies that all the elements coordinate properly and that the overall system functionality is achieved.

11.2.4 Strategic Issues

The best of testing strategies will fail if some issues are not looked into properly. These issues must be addressed to ensure successful software testing strategy.

- ***Specify product requirements in a quantifiable manner long before the testing begins:*** Testing assesses a lot of characteristics like portability, maintainability and usability other than just finding errors. Thus, these requirements must be specified in a quantifiable manner so that the testing results are unambiguous.
- ***Objectives of testing must be stated clearly:*** The testing objectives must be stated in measurable terms.
- ***Understand the software users and create a profile for each user category:*** Use-cases that describe the interaction of the classes of users can reduce overall testing efforts by concentrating on the actual use of the product.
- ***Develop a testing plan that emphasizes on rapid cycle testing:*** The testing must be done in cycles so that the feedback from each level can be used to control the level of quality and the corresponding test strategies.
- ***Use effective formal technical reviews as a filter prior to testing:*** FTRs cab be equally effective in finding errors. They reduce the overall testing time.
- ***Develop a continuous improvement approach for testing process:*** The test strategy should be measured. The metrics collected during testing process should be used as a part of statistical process control.

11.3 UNIT TESTING

Unit testing focuses on verification of individual units or modules of software. Using the design, important control paths are identified and tested to find errors within the

module boundary. The unit testing is white-box oriented and can be conducted in parallel for multiple components.

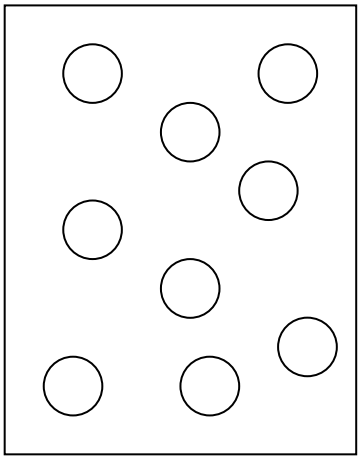


Figure 11.2: Unit Testing

It involves running a module in isolation from the rest of the software by preparing test cases and comparing the actual results with the expected results as specified by the specifications and design. One of its purposes is to find and remove as many errors in the software as practical.

11.3.1 Advantages of Unit Testing

- The size of a module is small enough to locate errors comparatively easily.
- The module is small enough to be able to test it in an exhaustive fashion.
- Confusing interactions of multiple errors in different parts of the software are eliminated.

However, there are problems associated with running a program in isolation. The biggest problem is how to run a module in isolation, without anything to call it and without anything being called by it. One approach is to build an appropriate routine to call it and the stubs to be called by it or to simply insert output statements.

These additional costs of writing code, called scaffolding, although include effort important to testing but are not delivered in the actual product, as shown in Figure 11.3.

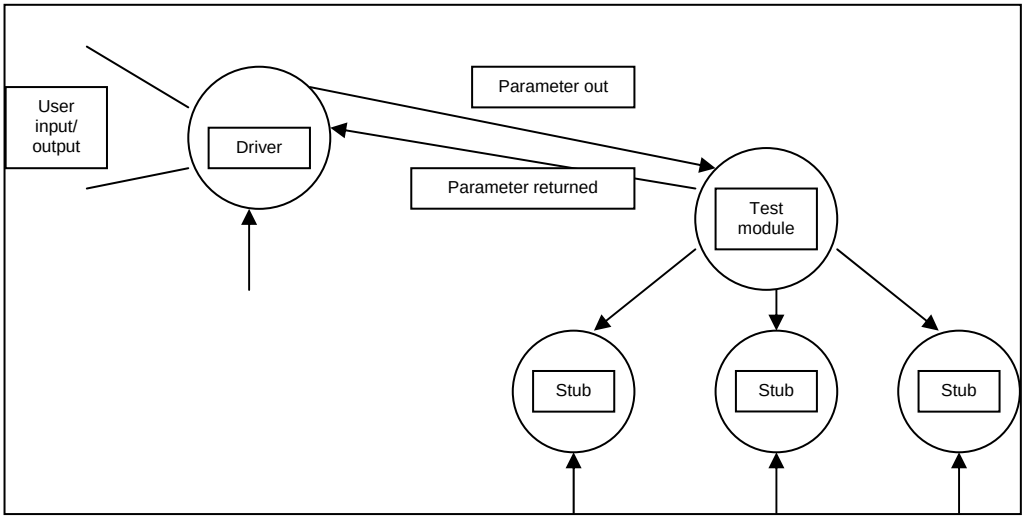


Figure 11.3: Scaffolding required to Test a Program Unit

Selective testing of execution paths: Test cases should be designed to detect maximum errors due to erroneous computations, incorrect comparisons or improper control flow. Basis path and loop testing are used to find a broad array of path errors.

Good design ensures that error conditions are anticipated and error-handling paths be set up to reroute or terminate processing when an error occurs. This approach is called anti-bugging.

The various errors that must be checked for while testing are as under:

a. Error description is unintelligible.

b. Error noted does not correspond to error encountered.

c. Error condition causes system intervention prior to error handling.

d. Exception-condition processing is incorrect.

e. Error description does not provide enough information to assist in the location of the cause of the error.

Boundary testing is the most important task of unit testing. Software often fails at boundaries.

Check Your Progress 1

Fill in the blanks:

1. Software often fails at .....
2. Confusing interactions of multiple errors in different parts of the software are eliminated using .....
3. ..... removes the conflict of interests that may be present otherwise.

11.4 INTEGRATION TESTING

Integration testing is the technique used for constructing the program structure and at the same time carrying out tests to find errors associated with interfacing. The objective is to build a program using the tested unit components as per design.

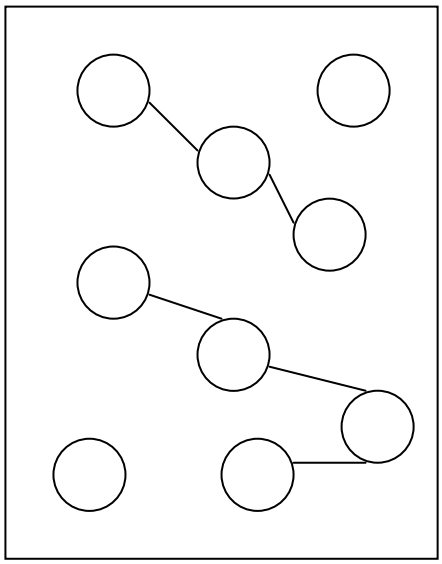


Figure 11.4: Integration Testing

There are several classical integration strategies like, top-down integration, bottom-up integration, regression testing, smoke testing, etc. These are discussed in detail here.

11.4.1 Top-down Integration

Top-down integration testing is an incremental approach towards to construct a program structure. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module. Modules subordinates to the main control module are included into the structure in either the depth-first or breadth first manner.

Depth-first integration integrates all components on a major control path of the structure. This selection of major path is arbitrary and depends on the application. Referring to Figure 11.5, the components C1, C2, C5 would be integrated first, followed by C8 or C6. Then, the central and right hand control paths are built. Breadth-first integration includes all the components that are direct subordinates at each level, moving across the structure horizontally. From the Figure 11.5, C2, C3, C4 would be integrated first followed by C5, C6, and so on.

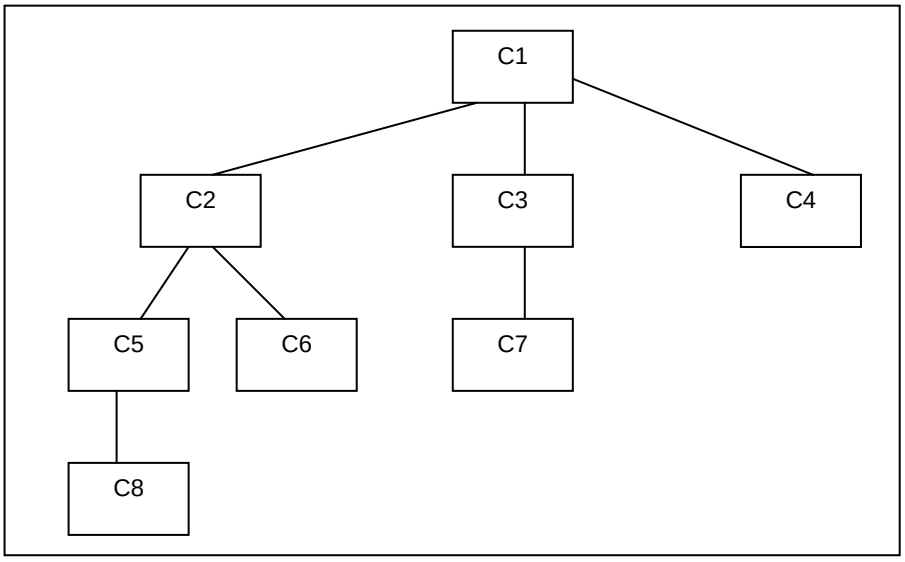


Figure 11.5: Top-down Integration

The integration step is performed in a series of five steps:

a. The main module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

b. Subordinate stubs are replaced by the actual components depending upon the integration approach.

c. Tests are conducted as each component is integrated.

d. On completing the tests for each set, another stub is replaced with the real component.

e. Regression testing may be conducted to ensure that new errors have not been introduced.

Top down strategy can lead to logical problems. The most common of these problems occurs when processing at low levels is required to test the higher levels.

11.4.2 Bottom-up Integration

Bottom-up Integration testing, begins construction with atomic modules. This technique eliminates the need for stubs as the components that have to be integrated are available for processing. It can be implemented using the following steps:

a. Low-level components are combined into clusters that perform a specific software sub-function.

b. A driver is written to coordinate test case input and output.

c. The cluster is tested.

d. Drivers are removed and clusters are combined moving upward in the program structure.

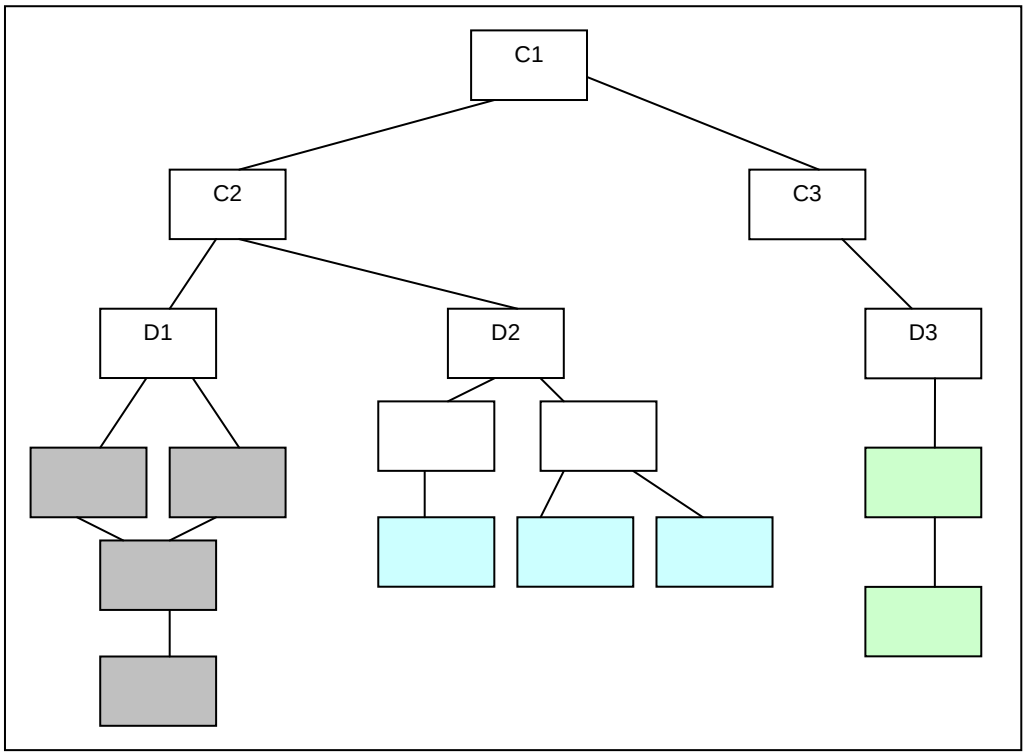


Figure 11.6: Bottom-up Integration

The various clusters in Figure 11.6 have been highlighted differently.

11.4.3 Regression Testing

Every time a new module is added, as a part of integration testing, the software changes. These changes may lead to problems with functions that were previously working fine. Thus, regression testing is the re-execution of some subset of tests that have been conducted already to ensure that changes have not lead to undesired side effects.

The regression test case contains three different classes of test cases:

- A representative sample of tests that will execute all software functions.
- Additional tests that focus on functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

Background

Experience has shown that as software is developed, this kind of reemergence of faults is quite common. Sometimes it occurs because a fix gets lost through poor revision control practices (or simple human error in revision control), but often a fix for a problem will be "fragile" in that it fixes the problem in the narrow case where it was first observed but not in more general cases which may arise over the lifetime of the software. Finally, it has often been the case that when some feature is redesigned, the same mistakes will be made in the redesign that were made in the original implementation of the feature.

Therefore, in most software development situations it is considered good practice that when a bug is located and fixed, a test that exposes the bug is recorded and regularly retested after subsequent changes to the program. Although this may be done through manual testing procedures using programming techniques, it is often done using automated testing tools. Such a test suite contains software tools that allow the testing environment to execute all the regression test cases automatically; some projects even set up automated systems to automatically re-run all regression tests at specified intervals and report any failures (which could imply a regression or an out-of-date test). Common strategies are to run such a system after every successful compile (for small projects), every night, or once a week. Those strategies can be automated by an external tool, such as BuildBot.

Regression testing is an integral part of the extreme programming software development method. In this method, design documents are replaced by extensive, repeatable, and automated testing of the entire software package at every stage in the software development cycle.

Traditionally, in the corporate world, regression testing has been performed by a software quality assurance team after the development team has completed work. However, defects found at this stage are the most costly to fix. This problem is being addressed by the rise of developer testing. Although developers have always written test cases as part of the development cycle, these test cases have generally been either functional tests or unit tests that verify only intended outcomes. Developer testing compels a developer to focus on unit testing and to include both positive and negative test cases.

Uses

Regression testing can be used not only for testing the correctness of a program, but often also for tracking the quality of its output. For instance, in the design of a compiler, regression testing should track the code size, simulation time and time of the test suite cases.

11.4.4 Smoke Testing

Smoke testing is an integration testing approach that is mainly used for testing shrink-wrapped software products. It is designed as a pacing mechanism for time-critical projects giving a chance for the project team to assess the software on a frequent basis. Thus, smoke testing includes the following activities:

- Software components that have been translated into code are clubbed together in the build. A build comprises of data files, libraries, reusable modules, etc.
- A series of tests will be defined to uncover the errors that are potential risks to the program functioning.
- The build is integrated with other builds and the entire product is smoke tested daily. This can be carried out using a top-down or bottom-up approach.

Smoke testing is done by developers before the build is released to the testers, or by testers before accepting a build for further testing. Microsoft claims that after code reviews, smoke testing is the most cost effective method for identifying and fixing defects in software.

In software engineering, a smoke test generally consists of a collection of tests that can be applied to a newly created or repaired computer program. Sometimes the tests are performed by the automated system that builds the final software. In this sense a smoke test is the process of validating code changes before the changes are checked into the larger product's official source code collection or the main branch of source code.

In software testing, a smoke test is a collection of written tests that are performed on a system prior to being accepted for further testing. This is also known as a build verification test. This is a "shallow and wide" approach to the application. The tester "touches" all areas of the application without getting too deep, looking for answers to basic questions like, "Can I launch the test item at all?", "Does it open to a window?", "Do the buttons on the window do things?".

The purpose is to determine whether or not the application is so badly broken that testing functionality in a more detailed way is unnecessary. These written tests can either be performed manually or using an automated tool. When automated tools are used, the tests are often initiated by the same process that generates the build itself. This is sometimes referred to as "rattle" testing - as in "if I shake it does it rattle?"

Thus, smoke testing can be defined as the testing carried out on the system from end-to-end. It need not be exhaustive but expose major problems. The smoke tests should be thorough enough that if the build passes, it can be assumed that the program is stable enough to be testes even more thoroughly.

The various benefits of smoke testing are as follows:

- ***Integration risk is minimized:*** As the smoke testing is done regularly and thoroughly, all the incompatibilities and other show-stopper errors are detected early thus reducing the serious impact when the errors are detected at later stages.
- ***The quality of the end-product is improved:*** Because the approach is integration oriented, it will uncover both functional and architectural errors and component-level design defects. Because of early defect correction, the end product is better in quality.
- ***Error diagnosis and correction are simplified:*** Errors detected during smoke testing are normally associated with new software increments. Thus, the error detection and correction is easier.
- ***Progress is easier to assess:*** Because each day more software is integrated and checked to work properly, it gives indication of the progress being made daily to the managers.

11.4.5 Documentation for Integration Testing

A test specification document contains the overall integration plan of the software and a description of specific test cases. This document contains the test plan and a test procedure, is a product of the software process and is a part of the software configuration.

The below mentioned criteria and corresponding test cases are applied for all test phases:

- ***Interface integrity:*** Internal and external interfaces are tested as each module is integrated into the structure.

- ***Functional integrity:*** Tests designed to detect functional errors are carried out.
- ***Information content:*** Tests designed to detect errors related to local or global data structures are carried out.
- ***Performance:*** Tests designed to verify the performance bounds identified during software design are carried out.

A history of test results, problems or peculiarities is recorded in the Test specification. This information is important during software maintenance.

Integration Testing Steps

Integration Testing typically involves the following Steps:

Step 1: Create a Test Plan

Step 2: Create Test Cases and Test Data

Step 3: If applicable create scripts to run test cases

Step 4: Once the components have been integrated execute the test cases

Step 5: Fix the bugs if any and re test the code

Step 6: Repeat the test cycle until the components have been successfully integrated

Integration Test Plan

As you may have read in the other articles in the series, this document typically describes one or more of the following:

- How the tests will be carried out
- The list of things to be Tested
- Roles and Responsibilities
- Prerequisites to begin Testing
- Test Environment
- Assumptions
- What to do after a test is successfully carried out
- What to do if test fails
- Glossary

Writing an Integration Test Case

Simply put, a Test Case describes exactly how the test should be carried out.

The Integration test cases specifically focus on the flow of data/information/control from one component to the other.

So the Integration Test cases should typically focus on scenarios where one component is being called from another. Also the overall application functionality should be tested to make sure the app works when the different components are brought together.

The various Integration Test Cases clubbed together form an Integration Test Suite

Each suite may have a particular focus. In other words different Test Suites may be created to focus on different areas of the application.

As mentioned before a dedicated Testing Team may be created to execute the Integration test cases. Therefore the Integration Test Cases should be as detailed as possible.

Sample Test Case Table

Test Case ID	Test Case Description	Input Data	Expected Result	Actual Result	Pass/Fail	Remarks

Additionally the following information may also be captured:

a) Test Suite Name

b) Tested By

c) Date

d) Test Iteration (One or more iterations of Integration testing may be performed)

Working towards Effective Integration Testing

There are various factors that affect Software Integration and hence Integration Testing:

1. ***Software Configuration Management:*** Since Integration Testing focuses on Integration of components and components can be built by different developers and even different development teams, it is important the right version of components are tested. This may sound very basic, but the biggest problem faced in n-tier development is integrating the right version of components. Integration testing may run through several iterations and to fix bugs components may undergo changes. Hence it is important that a good Software Configuration Management (SCM) policy is in place. We should be able to track the components and their versions. So each time we integrate the application components we know exactly what versions go into the build process.
2. ***Automate Build Process where Necessary:*** A lot of errors occur because the wrong version of components were sent for the build or there are missing components. If possible write a script to integrate and deploy the components this helps reduce manual errors.
3. ***Document:*** Document the Integration process/build process to help eliminate the errors of omission or oversight. It is possible that the person responsible for integrating the components forgets to run a required script and the Integration Testing will not yield correct results.
4. ***Defect Tracking:*** Integration Testing will lose its edge if the defects are not tracked correctly. Each defect should be documented and tracked. Information should be captured as to how the defect was fixed. This is valuable information. It can help in future integration and deployment processes.

11.5 VALIDATION TESTING

After the software has been integration tested and all the interface errors have been detected and fixed, a final round of testing is carried out, it is validation testing. Validation succeeds when software functions the way the customer expects it to. Validation specification contains the relevant information required to carry out validation testing.

Validation testing is carried out by performing a series of black-box tests that ensure that the software is in conformance with the requirements. Both the plan and procedure are defined to ensure that all the behavioral, functional, procedural, performance, documentation, etc. is correct and all requirements have been met.

After the validation tests are carried out, it may lead to two possibilities: (1) the function is in agreement with the specification and are acceptable or (2) there exists some deviations from the specification and deficiency list is created. These deficiencies need some time to get fixed and have to be discussed with the customer.

11.5.1 Configuration Review

Configuration review is an important part of validation testing. Its main aim is to ensure that all elements of software configuration are properly developed and cataloged. It is also called audit.

Software audit can mean:

- a software licensing audit, where a user of software is audited for licence compliance
- software quality assurance, where a piece of software is audited for quality
- a software audit review, where a group of people external to a software development organisation examines a software product
- a physical configuration audit
- a functional configuration audit

Objectives and Participants

"The purpose of a software audit is to provide an independent evaluation of conformance of software products and processes to applicable regulations, standards, guidelines, plans, and procedures". The following roles are recommended:

- The Initiator (who might be a manager in the audited organization, a customer or user representative of the audited organization, or a third party), decides upon the need for an audit, establishes its purpose and scope, specifies the evaluation criteria, identifies the audit personnel, decides what follow-up actions will be required, and distributes the audit report.
- The Lead Auditor (who must be someone "free from bias and influence that could reduce his ability to make independent, objective evaluations") is responsible for administrative tasks such as preparing the audit plan and assembling and managing the audit team, and for ensuring that the audit meets its objectives.
- The Recorder documents anomalies, action items, decisions, and recommendations made by the audit team.
- The Auditors (who must be, like the Lead Auditor, free from bias) examine products defined in the audit plan, document their observations, and recommend corrective actions. (There may be only a single auditor.)
- The Audited Organization provides a liaison to the auditors, and provides all information requested by the auditors. When the audit is completed, the audited organization should implement corrective actions and recommendations.

11.5.2 Alpha and Beta Testing

When custom software is built for a customer, a series of acceptance tests are carried out to enable the customer to validate all the requirements. These tests are conducted

by the end user right from an informal manner to a systematic and planned series of tests. This may continue over a period of weeks or months. Most software builders use a process called alpha and beta testing to detect errors that only the end-user is able to find.

The alpha test is carried out at the developer's site by the customer. The software engineer, here records all the errors and usage problems. These tests are conducted in a controlled environment.

The beta tests are carried out at more than one customer sites by the end users of the software. The developer is not normally present for this type of testing activities. Thus, it is actually a live testing of the software in an environment that is not controlled by the developer. All the problems are recorded by the customer and are reported to the developer. Beta testing follows the alpha testing phase.

11.6 SYSTEM TESTING

System testing is a means to carry out a series of tests whose primary purpose is to fully exercise the computer based system. The tests are carried out to ensure that all system elements have been integrated properly and perform the desired functions as allocated. We will discuss the various types of system testing techniques in detail here.

System testing is black box testing, performed by the Test Team, and at the start of the system testing the complete system is configured in a controlled environment. The purpose of system testing is to validate an application's accuracy and completeness in performing the functions as designed. System testing simulates real life scenarios that occur in a "simulated real life" test environment and test all functions of the system that are required in real life. System testing is deemed complete when actual results and expected results are either in line or differences are explainable or acceptable, based on client input.

Upon completion of integration testing, system testing is started. Before system testing, all unit and integration test results are reviewed by SWQA to ensure all problems have been resolved. For a higher level of testing it is important to understand unresolved problems that originate at unit and integration test levels.

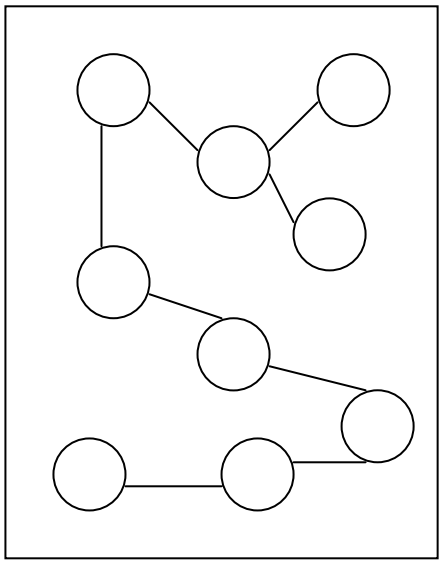


Figure 11.7: System Testing

11.6.1 Recovery Testing

Recovery testing is a system testing technique the causes the software to fail because of various reasons and verify that the recovery is being properly performed. In case of an automatic recovery, operations like check-pointing, re-initialization, data recovery and restart are checked for correctness. If the recovery requires human intervention, the mean-time-to-repair (MTR) is calculated to determine whether it is within the accepted limits.

Recovery testing is basically done in order to check how fast and better the application can recover against any type of crash or hardware failure etc. Type or extent of recovery is specified in the requirement specifications. It is basically testing how well a system recovers from crashes, hardware failures, or other catastrophic problems

11.6.2 Security Testing

Security testing is used to verify that the protection mechanism built into the system is capable enough to protect it from improper penetration. The system security must be tested for vulnerability from both frontal and rear attack.

During security testing the tester pays the role of an individual who wishes to enter the system by acquiring passwords, may attack the software with customized software to break into the system security, etc.

A good design must ensure that the penetration cost is more than the value of information obtained.

Security testing has recently moved beyond the realm of network port scanning to include probing software behavior as a critical aspect of system behavior. Unfortunately, testing software security is a commonly misunderstood task.

11.6.3 Stress Testing

Stress testing executes the system in a manner that requires resources in abnormal quantity, frequency or volume by designing test cases that generate multiple interrupts per second, which require maximum memory and other resources, which may cause the thrashing of the virtual operating system, etc.

A variation of stress testing is sensitivity testing. In certain cases a small range of data contained within the bounds of valid data for a program may cause severe errors and performance degradation. It attempts to detect the data combinations within valid input classes that may cause instability or incorrect processing.

Stress testing deals with the quality of the application in the environment. The idea is to create an environment more demanding of the application than the application would experience under normal workloads. This is the hardest and most complex category of testing to accomplish and it requires a joint effort from all teams.

A test environment is established with many testing stations. At each station, a script is exercising the system. These scripts are usually based on the regression suite. More and more stations are added, all simultaneous hammering on the system, until the system breaks. The system is repaired and the stress test is repeated until a level of stress is reached that is higher than expected to be present at a customer site.

Race conditions and memory leaks are often found under stress testing. A race condition is a conflict between at least two tests. Each test works correctly when done in isolation. When the two tests are run in parallel, one or both of the tests fail. This is usually due to an incorrectly managed lock.

A memory leak happens when a test leaves allocated memory behind and does not correctly return the memory to the memory allocation scheme. The test seems to run correctly, but after being exercised several times, available memory is reduced until the system fails.

11.6.4 Performance Testing

Performance testing is designed to test the run-time performance of software after it has been integrated. It occurs throughout all steps of the integrated testing. Performance testing at the module level can also be carried out during white-box testing.

Performance tests are at time coupled with stress testing and require both software and hardware instrumentation, in order to measure resource utilization.

Performance Testing assesses an application on speed, scalability and stability. Essentially, it gauges the application's response to user requests under expected circumstances and determines its stability and limits under stressful situations.

In simple terms, performance tests are designed to simulate a particular workload. The workload is defined as the total burden of activity placed on the application. This burden consists of a number of virtual users who process a defined set of transactions in a specified time period. Assigning the proper workload is one of the most crucial parts of any performance analysis. Testing should be conducted to assess performance for three workload categories:

- ***Steady state:*** A constant number of users using the application for the entire duration of the test; this primarily examines the application for stress and stability.
- ***Increasing workload:*** Used to assess the maximum user load the application can take; virtual users are added in steps until the application no longer responds.
- ***Scenario-based:*** This is designed based on the business users' inputs on typical workloads in a day and is the most realistic test.

Types of Performance Testing

Different types of performance testing must be planned and executed for an application/product depending on scope/need.

- Load testing helps to ensure the system performs per requirements under the anticipated load level for the purpose of identifying problems in resource contention, response times. It also determines the minimum configuration under which the system can perform satisfactorily.
- Stress testing probes the behavior of the application under very heavy load to determine its capacity limits or to identify limits imposed by the product design or its environment.
- Endurance testing uncovers issues pertaining to non-release of system resources can be identified only when it is run over long periods of time with normal user loads. Endurance Testing helps assess the application's stability over long periods of time.

Test Approach

A best practice test approach begins in the development process and is part of the application's planning and strategy. The active phases of testing include design, execution and reporting.

Design: In the design phase, test scenarios are drafted based on user inputs on transaction volume, typical usage pattern of the application and typical user loads accessing parts of the application.

Each of the high-level scenarios consists of a number of different functional (transaction) operations to be executed by varying number of users at the same point in time. This would, more or less, reflect the real-time usage of the system in production.

Tool-based execution: The high-level scenarios are translated into test cases and written into test scripts for execution by a performance-testing tool. Rather than requiring real users to execute scenarios in tandem, the tool would facilitate creation of virtual users without requiring the same hardware base.

A tool enables simulation of test runs for various workload models emulating real-time user roles and access patterns. Test runs can be controlled as desired and even performed for an extended duration of time. Further, tools also provide monitoring logs and performance metrics both as reports and graphs for easy interpretation and analysis.

Reports and Metrics: Performance metrics include statistics on system resources, performance (%CPU time, interrupts, etc.), memory, logical disc operations, and network interface details. Standard performance testing tools are capable of producing a variety of reports and graphs to show measurements of performance.

Check Your Progress 2

State whether the following statements are true or false:

1. Performance testing is designed to test the run-time performance of software before it has been integrated.
2. The alpha test is carried out at the developer's site by the developer.
3. Validation succeeds when software functions the way the customer expects it to.

11.7 DEBUGGING

The goal of testing is to identify errors in the program. The process of testing gives symptoms of the presence of an error. After getting the symptom, we begin to investigate to localize the error i.e. to find out the module causing the error. This section of code is then studied to find the cause of the problem. This process is called debugging. Hence, debugging is the activity of locating and correcting errors. It starts once a failure is detected.

11.7.1 Debugging Techniques

Although developers learn a lot about debugging from their experiences, but these are applied in a trial and error manner. Debugging is not an easy process as per human psychology as error removal requires the acceptance of error and an open mind willing to admit the error.

However, Pressman explains certain characteristics of bugs that can be useful for developing a systematic approach towards debugging. These are:

- The symptoms and cause may be geographically remote. That is, the symptom may appear in one part of the program, while the cause may actually be located in other part. Highly coupled programs can worsen this situation.
- The symptom may disappear (temporarily) when another error is corrected.

- The symptom may actually be caused by non errors (e.g. round off inaccuracies).
- The symptom may be caused by a human error that is not easily traced.
- The symptom may be result of timing problems rather than processing problems.
- It may be difficult to accurately reproduce input conditions (e.g. a real time application in which input ordering is indeterminate).
- The symptom may be intermittent. This is particularly common in embedded systems that couple hardware with software inextricably.
- The symptom may be due to causes that are distributed across a number of tasks running on different processors.

11.7.2 Debugging Approaches

The debugging approach can be categorized into various categories. The first one is trial and error. The debugger looks at the symptoms of the errors and tries to figure out that from exactly which part of the code the error originated. Once found the cause, the developer fixes it. However, this approach is very slow and a lot of time and effort goes waste.

The other approach is called backtracking. Backtracking means to examine the error symptoms to see where they were observed first. One then backtracks in the program flow of control to a point where these symptoms have disappeared. This process identifies the range of the program code which might contain the cause of errors. Another variant of backtracking is forward tracking, where print statements or other means are used to examine a sequence of intermediate results to determine the point at which the result first becomes wrong.

The third approach is to insert watch points (output statements) at the appropriate places in the program. This can be done using software without manually modifying the code.

The fourth approach is more general and called induction and deduction.

The induction approach comes from the formulation of a single working hypothesis based on the data, analysis of the existing data and on especially collected data to prove or disprove the working hypothesis. The inductive approach is explained in Figure 11.8.

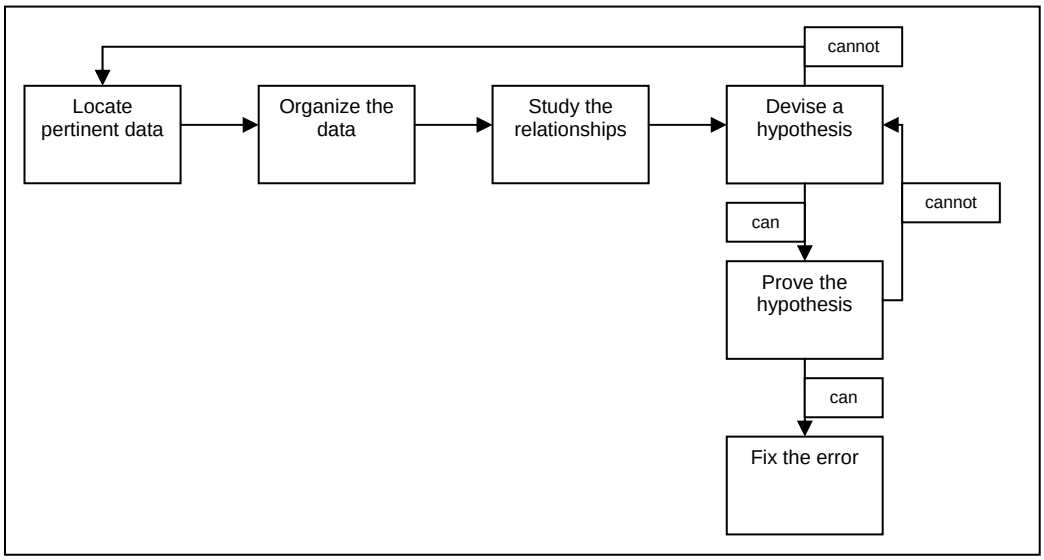


Figure 11.8: Inductive Debugging Approach

The deduction approach begins by enumerating all causes or hypothesis, which seem possible. Then, one by one, particular causes are ruled out until a single one remains for validation.

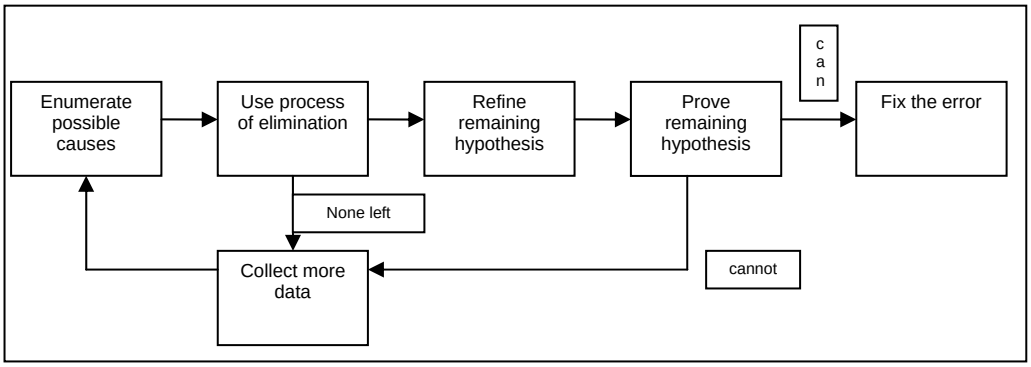


Figure 11.9: Deductive Debugging Approach

11.8 LET US SUM UP

Software testing accounts for the largest share of total technical efforts put in the software process. Its objective is to uncover errors. In order to fulfill this objective, a series of test steps are planned and executed – unit, integration, validation and system testing.

Unit and integration tests concentrate on functional verification and then integrating these components together into a program structure. Validation tests ensure traceability to requirements and system testing validates software once it has been incorporated into a larger system. Unlike testing, debugging can be considered as an art of resolving errors encountered while testing. Starting with the symptoms' identification it must track down to the cause of error.

11.9 LESSON END ACTIVITIES

1. How do design attributes facilitate debugging?
2. List the various problems that can come up as a result of adding debugging statements into the code.
3. Is unit testing necessary in all scenarios? Provide examples to justify your answer.

11.10 KEYWORDS

MTR: Mean-Time-to-Repair

FTR: Formal Technical Reviews

SQA: Software Quality Assurance

ITG: Independent Test Group

Verification: Verification refers to the set of activities that ensure that the software correctly implements a specific function.

Validation: Validation refers to a different set of activities that ensure that the software that has been built is traceable to the customer requirements.

Unit testing: Unit testing begins at the vortex of the spiral and lays emphasis on each unit of the software as implemented in the source code.

Anti-bugging: The process of anticipating errors and setting up error-handling paths to reroute or terminate processing when an error occurs is called anti-bugging.

Integration testing: Integration testing is the technique used for constructing the program structure and at the same time carrying out tests to find errors associated with interfacing.

Smoke testing: Smoke testing is an integration testing approach that is mainly used for testing shrink-wrapped software products.

Configuration review: A review carried out to ensure that all elements of software configuration are properly developed and cataloged.

System testing: System testing is a means to carry out a series of tests whose primary purpose is to fully exercise the computer based system.

Recovery testing: Recovery testing is a system testing technique the causes the software to fail because of various reasons and verify that the recovery is being properly performed.

Security testing: Security testing is used to verify that the protection mechanism built into the system is capable enough to protect it from improper penetration.

Performance testing: Performance testing is designed to test the run-time performance of software after it has been integrated.

11.11 QUESTIONS FOR DISCUSSION

1. What is the difference between system testing and validation testing?
2. Does exhaustive testing ensure 100% defect free software?
3 Discuss the various software testing strategies in detail.

Check Your Progress: Model Answers

CYP 1

1. Boundaries
2. Unit testing
3. Independent testing

CYP 2

1. False
2. False
3. True

11.12 SUGGESTED READINGS

Boris Beizer, *Software Testing Techniques*, Second Edition, Dreamtech Press, 2003.

Myers and Glenford, J., *The Art of Software Testing*, John-Wiley & Sons, 1979.

Roger, S. Pressman, *Software Engineering – A Practitioner's Approach*, 5th Edition, McGraw Hill, 2001.

Marnie, L. Hutcheson, *Software Testing Fundamentals*, Wiley-India, 2007.

MODEL QUESTION PAPER

MCA

Third Year

Sub: Software Testing

Time: 3 hours **Total Marks:** 100

Direction: There are total eight questions, each carrying 20 marks. You have to attempt any five questions.

---

1. Is complete testing possible? Justify with proper reasoning.
2. How can we measure the importance of a bug? Give a mathematical formula to measure the same.
3. Discuss the difference between worst case and ad-hoc test case performance evaluation method of testing.
4. What are predicates, path predicates and achievable paths? Explain the relation between all of them with respect to software testing.
5. Explain the role of walkthroughs and reviews in the transaction based testing.
6. Compare the effectiveness of the various strategies for data-flow testing.
7. Describe transition bugs giving reference to the unreachable and dead states.
8. What is client-server architecture? What are the problems encountered while testing these applications?