



KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Eachanari (po), Coimbatore-21

DEPARTMENT OF CS, CA & IT

SUBJECT NAME: SOFTWARE TESTING

SUBJECT CODE: 17ITU501B

SEMESTER: VI

STAFF: G.MANIVASAGAM

CLASS: III B.Sc. CT

SCOPE

Software Testing is designed to establish that the software is working satisfactorily as per the requirements.

COURSE OBJECTIVES

- Software Testing is a process designed to prove that the program is error free.
- Software The job of testing is to certify that the software does its job correctly and can be used in production.

UNIT – I Testing Fundamentals

Examining the Specification: Getting started – Performing a high-level review of the specification – Low-level specification test techniques. Testing the software with blinders on: Dynamic Black-Box Testing- Test-to-Pass and Test-to-Fail- Equivalence Partitioning- Data testing – State testing – Other Black-box test techniques.

UNIT – II Examining the code

Static White-Box testing- Formal reviews – Coding Standards and Guidelines- Generic Code Review Checklist. Testing the software with X-Ray glasses: Dynamic White-Box testing- Dynamic White-Box testing versus Debugging-Testing the Pieces- Data Coverage- Code Coverage.

Flowgraphs 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 and Data-Flow Testing

Transaction Flows-Transaction Flow Testing Techniques. Data-Flow Testing Basics- Data-Flow Testing Strategies-Application, Tools, Effectiveness

UNIT – IV Domain Testing

Domains and Paths-Domain Testing-Domains and Interface Testing-Domains and Testability

UNIT – V Logic-Based Testing and State Graphs

Motivational Overview-Decision Tables-Path Expressions Again-KV Charts-Specifications
State Graphs-Good State Graphs and Bad-State Testing

Suggested Readings

1. Boris Beizer (2009), Software Testing Techniques (2nd ed.). New Delhi Dreamtech Press
2. Ron Patton (2002) Software Testing (2nd ed.). New Delhi: Pearson Education
3. Dorothy Graham, Erik Van Veenendaal, Isabel Evans, Rex Black (2007). Foundations of Software Testing, ISTQB Certification.
4. Brian Hambling, Peter Morgan, Angelina Samaroo, Geoff Thompson (2010). Software Testing , (2nd ed.). An ISEB Foundation, BCS
5. Renu Rajani, Pradeep Oak (2004). Software Testing- Effective Methods, Tools and Techniques, Tata McGraw Hill, New Delhi

Web Sites

1. www.testinggeek.com
2. www.softwaretestinghelp.com
3. www.softwaretestinginstitute.com



KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Eachanari (po), Coimbatore-21

DEPARTMENT OF CS, CA & IT

SUBJECT NAME: SOFTWARE TESTING

SUBJECT CODE: 17CTU501B

SEMESTER: VI

STAFF: G.MANIVASAGAM

CLASS: III B.Sc. CT

S.No	Lecture Duration (Hr)	Topics to be Covered	Support Materials
Unit I			
1.	1	Software Testing Fundamentals - Examining the Specification Getting started Black box and White box testing	R1: 53-57 W1
2.	1	Static and Dynamic testing	
3.	1	Static Black-box testing	
4.	1	Performing a High-level Review of Specification Test Techniques	R1: 57-62 W1
5.	1	Low-level Specification Test Techniques	R1: 57-62 W1
6.	1	Testing the software with Blinders on Dynamic Black-Box Testing	R1: 63-69 W1
7.	1	Test-to-pass and Test-to-fail	R1: 63-69 W1
8.	1	Equivalence Partitioning	R1: 63-69 W1
9.	1	Data Testing	R1: 70-79 W1
10	1	State testing	R1: 79-87 W1

11	1	Other Black-Box test Techniques	R1: 87-95 W1
12	1	Recapitulation and discussion of important questions	
Total No. of Hours Planned for Unit I			9
Unit II			
1.	1	Examining the code Static White-Box testing Formal reviews Coding Standards and Guidelines Generic Code Review Checklist	R1: 87-95 R1: 96-104 W1
2.	1	Testing the software with X-Ray glasses Dynamic White-Box testing	R1: 105-108
3.	1	Dynamic White-Box testing versus Debugging Testing the Pieces Data Coverage , Code Coverage.	R1: 108-113 R1: 113-116 R1: 117-121 W1
4.	1	Flowgraphs and Path Testing Path-testing Basics Motivation and Assumptions Control Flowgraphs	S1: 59-70
5.	1	Predicates, Path Predicates and Achievable Paths General Predicates Predicate Expressions	S1: 92 - 98
6.	1	Path Sensitizing Review; Achievable and Unachievable Paths Pragmatic Observations Heuristic Procedures for sensitizing Paths Examples	S1: 101-109
7.	1	Path Instrumentation The Problem General Strategy Link Markers Link Counters Other Instrumentation Methods	S1: 109-115

		Implementation	
8.	1	Implement and Application of Path Testing Integration, Coverage and Paths in Called Components New code Maintenance Rehosting	S1: 115-118
9.	1	Recapitulation and discussion of important questions	
Total No. of Hours Planned for Unit II			9
Unit III			
1.	1	Transaction Flow Testing Transaction Flows Definitions Example Usage Implementation	S1: 122-128
2.	1	Perspective Complications Transaction Flow Structure	S1: 122-128
3.	1	Transaction flow Testing Techniques Get the Transaction Flows Inspection Reviews and Walkthroughs Path Selection	S1: 128 - 136 R1: 672-673
4.	1	Sensitization Instrumentation Test Databases Execution	S1: 136-143
5.	1	Data Testing Data Flow Testing Basics Motivation and Assumptions	S1: 145-150 R1: 105
		Data Flowgraphs	S1: 150-157
6.	1	The Data-Flow model	S1: 157-161
7.	1	Data-Flow Testing Strategies General Terminology	S1: 161-167 R1: 238-239
8.	1	The Strategies Slicing, Dicing, Data Flow and Debugging	S1: 161-167 R1: 238-239

		Application, Tools, Effectiveness	S1: 167-171
9.	1	Recapitulation and discussion of important questions	
Total No. of Hours Planned for Unit III			9
Unit IV			
1.	1	Domain Testing Domains and Paths The Model A Domain Is a Set Domains, Paths and Predicates	S1:173-176 R1: 241
2.	1	Domain Closure Domain Dimensionality	S1: 176-182
3.	1	The Bug Assumptions Restrictions	S1: 176-182
4.	1	Domain Testing Overview Domain Bugs and How to Test for them	S1: 192-195
5.	1	Procedure Variations, Tools, Effectiveness	S1: 195-202
6.	1	Domains and Interface Testing General Domains and Range	S1: 202 – 207 R1: 236
7.	1	Closure Compatibility Span Compatibility	S1: 202 – 207 R1: 236
		Interface Range/Domain Compatibility Testing Finding the values	S1: 202 – 207 R1: 236
8.	1	Domains and Testability General Linearizing Transformations Coordinate Transformations	S1: 207 - 211
		A Canonical Program Form Great Insights?	S1: 207 - 211
9.	1	Recapitulation and discussion of important questions	
Total No. of Hours Planned for Unit IV			9
Unit V			
1.	1	Motivational Overview Programmers and Logic Hardware and Logic Testing	S1: 320 – 326 R1: 660

		Specification Systems and Languages Knowledge-Based Systems Overview	
2.	1	Decision Tables Definitions and Notation Decision-Table Processors Decision Tables as a Basis for Test Case Design Expansion of Immaterial Cases Test Case Design Decision Tables and structure	R1: 180 S1: 326 – 332
3.	1	Path Expression Again General Boolean Algebra Boolean Equations	S1: 332 – 334 S1: 334 – 343
4.	1	KV Charts The Problem Simple Forms Three Variables Four Variables and More Even More Testing Strategies	S1: 343 – 347 S1: 347 – 352
5.	1	Specifications General Finding and Translating the Logic Ambiguities and Contradictions Don't Care and Impossible Terms	S1: 352 – 355 S1: 355 – 359
6.	1	State Graphs State Graphs States Inputs and Transitions Outputs	S1: 364 – 373 R1: 238
7.	1	Good State Graphs and Bad General State Bugs Transition Bugs Output Errors Encoding Bugs	S1: 373 – 380 S1: 380 – 387
8.	1	State Testing Impact of Bugs Principles Limitations and Extensions What to Model Getting the Data Tools	S1: 387 – 391
9.	1	Recapitulation and Discussion of Important Questions	

10.	1	Discussion of Previous ESE Question Papers	
11.	1	Discussion of Previous ESE Question Papers	
12.	1	Discussion of Previous ESE Question Papers	
Total No. of Hours Planned for Unit V			12
Total No. of Hours: 48			

Suggested Readings

S1: Ron Patton, 2002 Software Testing, 2nd Edition, New Delhi: Pearson Education, New Delhi.

S2: Boris Beizer, 2003, Software Testing Techniques, 2nd Edition, Dreamtech Press, New Delhi

References

R1: William E. Perry, 2001 Effective methods for Software Testing, 2nd Edition, New Delhi: John Wiley & Sons, Inc.,

R2: Dorothy Graham, Erik Van Veenendaal, Isabel Evans, Rex Black, 2007, Foundations of Software Testing, ISTQB Certification.

R3: Brian Hambling, Peter Morgan, Angelina Samaroo, Geoff Thompson, 2010, Software Testing, 2nd Edition, an ISEB Foundation, BCS

R4: Renu Rajani, Pradeep Oak, 2004, Software Testing- Effective Methods, Tools and Techniques, Tata McGraw Hill, New Delhi

Web Sites

W1: www.testinggeek.com

W2: www.softwaretestinghelp.com

W3: www.softwaretestinginstitute.com

W4: www.effectivesoft.com

W5: www.softwaresucks.com

UNIT-I
SYLLABUS

Examining the Specification: Getting started – Performing a high-level review of the specification – Low-level specification test techniques. Testing the software with blinders on: Dynamic Black-Box Testing- Test-to-Pass and Test-to-Fail- Equivalence Partitioning- Data testing – State testing – Other Black-box test techniques.

EXAMINING THE SPECIFICATION

1.1 Getting started

"The Software Development Process" involves the following models big-bang, code-and-fix, waterfall, and spiral. In each of the software development model, except big-bang, the development team creates a product specification from the requirements document to define what the software will become. The product specification is a written document using words and pictures to describe the intended product.

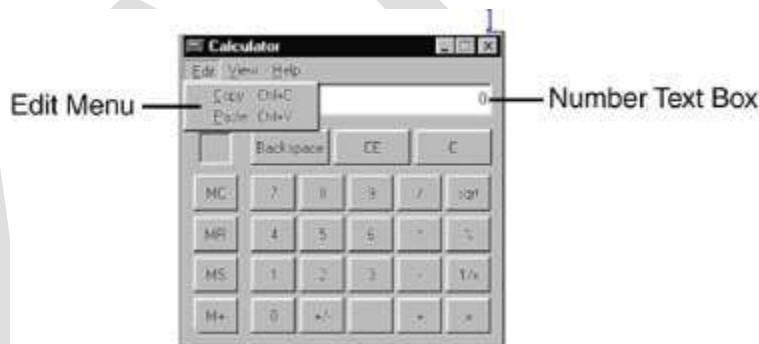


Figure 2.1 Standard Windows Calculator displaying the drop-down Edit menu.

In a windows calculator the product specification might be something like this

- The EDIT menu will have two selections Copy and Paste. These can be chosen by one of the three methods: pointing and clicking to the menu items with the mouse, using access keys(Alt+E and then C for Copy and V for Paste), or using the standard windows shortcut keys of Ctrl+C for Copy and Ctrl+V for paste

-
- The copy function will copy the current entry displayed in the number text box into the Windows Clipboard. The paste function will paste the value stored in the Windows Clipboard into the number text box.
 - The only way to assure that the end product is what the customer required and to properly plan the test effort is to thoroughly describe the product in a specification.
 - The other advantage of having a detailed specification is that a tester will have a document as a testable item and it can be used to find bugs before the first line of code is written

Black box and White box testing

Two terms that software testers use to describe the approach of testing are black-box testing and white-box testing. [Figure 2.2](#) shows the difference between the two approaches. In black-box testing, the tester only knows what the software is supposed to do he can't look in the box to see how it operates. If he types in a certain input, he gets a certain output. He doesn't know how or why it happens, just that it does.

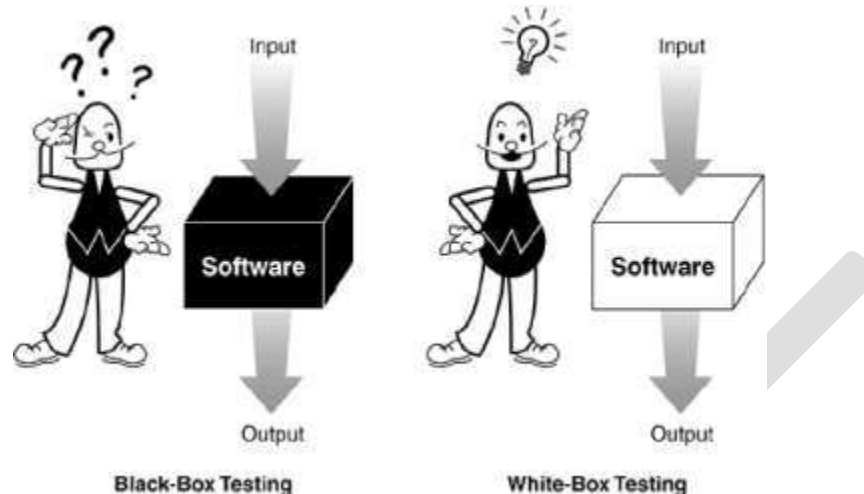


Figure 2.2 Black-box testing and White box testing

Black box Testing

- Black-box testing is sometimes referred to as functional testing or behavioral testing.
- In Windows Calculator shown in [Figure 2.1](#), if a number 3.14159 is typed and the SQRT button is pressed, the result is displayed as 1.772453102341
- With black-box testing, it doesn't matter what gyrations the software goes through to compute the square root of pi. It just does it.
- As a software tester, the result on "certified" calculator is only verified and determined if the Windows Calculator is functioning correctly.

White box Testing

- White-box testing is sometimes called clear-box testing
- The software tester has access to the program's code and can examine it for clues to help him with his testing.
- He can see inside the box
- Based on what he sees, the tester may determine that certain numbers are more or less likely to fail and can tailor his testing based on that information.

- There is a risk to white-box testing. It's very easy to become biased and fail to objectively test the software because the tester may tailor the tests to match the code's operation.

Static and Dynamic Testing

Two other terms used to describe method software is tested are static testing and dynamic testing. Static testing refers to testing something that's not running that is examining and reviewing it. Dynamic testing is that software is tested while running and using the software. The best analogy for these terms is the process you go through when checking out a used car. Kicking the tires, checking the paint, and looking under the hood are static testing techniques. Starting it up, listening to the engine and driving down the road are dynamic testing techniques.

Static Black Box Testing

- It is a method for testing the Specification. Specification is represented as a document but not an executing program.
- A specification is created using data from many sources.
- There is no need to get the reason why that information was collected.
- Just take the document perform static black box testing and examine for bugs.
- Specification may not be always in text format. Its format may be drawings also.
- If there is no specification found then source may be taken from developer, project manager or marketer. Then record and circulate it for review

1.2 Performing High Level Review of Specification

- While testing don't jump straight and look for bugs in code.
- Stand back and view it from high level.
- If there is better understanding of why and how then examination will be in detail.

1. Pretend to be a customer:

- Get known about end user.
- Understand customer expectation.
- Don't assume anything to be correct.
- If bugs are found, it is better.
- Test security of the software also.

2. Research existing standards and guidelines

- Back in days every software structure of every company like Microsoft and Apple are different. So it requires retraining.
- Now all software and hardware are standardized. So products are similar in look and feel.
- Standard should be strictly adhered
- Specifying Guidelines is optional but should be followed.

Example of standards and guidelines:

- Corporate terminology and conventions.
- Industry requirements.
- Government standards.
- Graphical User Interface
- Security standards.

Tester – It defines guidelines and standards applied to product developed.
A tester tests if standards are used and not overlooked.

3. Review and test similar software

The following are the things to be looked when reviewing competitive product.

- Scale – features included
- Complexity
- Testability
- Quality / Reliability
- Security

Read online and printed software reviews and articles about competitor.

1.3 Low Level Specification Test technique

- Testing specification at lower level.

Specification attributes checklists – The following attributes must be verified

1. Complete. Is anything missing or forgotten? Is it thorough? Does it include everything necessary to make it stand alone?
2. Accurate. Is the proposed solution correct? Does it properly define the goal? Are there any errors?
3. Precise, Unambiguous, and Clear. Is the description exact and not vague? Is there a single interpretation? Is it easy to read and understand?
4. Consistent. Is the description of the feature written so that it doesn't conflict with itself or other items in the specification?
5. Relevant. Is the statement necessary to specify the feature? Is it extra information that should be left out? Is the feature traceable to an original customer need?

6. Feasible. Can the feature be implemented with the available personnel, tools, and resources within the specified budget and schedule?
7. Code-free. Does the specification stick with defining the product and not the underlying software design, architecture, and code?
8. Testable. Can the feature be tested? Is enough information provided that a tester could create tests to verify its operation?

Specification Terminology Characteristics

- Always, Every, All, None, Never. If these words are seen such as these that denote something as certain or absolute, make sure that it is, indeed, certain.
- Certainly, Therefore, Clearly, Obviously, Evidently. These words tend to persuade you into accepting something as a given. Don't fall into the trap.
- Some, Sometimes, Often, Usually, Ordinarily, Customarily, Most, Mostly. These words are too vague. It's impossible to test a feature that operates "sometimes."
- Etc., And So Forth, And So On, Such As. Lists that finish with words such as these aren't testable. Lists need to be absolute or explained so that there's no confusion as to how the series is generated and what appears next in the list.
- Good, Fast, Cheap, Efficient, Small, Stable. These are unquantifiable terms. They aren't testable. If they appear in a specification, they must be further defined to explain exactly what they mean.
- Handled, Processed, Rejected, Skipped, Eliminated. These terms can hide large amounts of functionality that need to be specified.
- If...Then (but missing Else). Look for statements that have "If...Then" clauses but don't have a matching "Else." Ask yourself what will happen if the "if" doesn't happen.

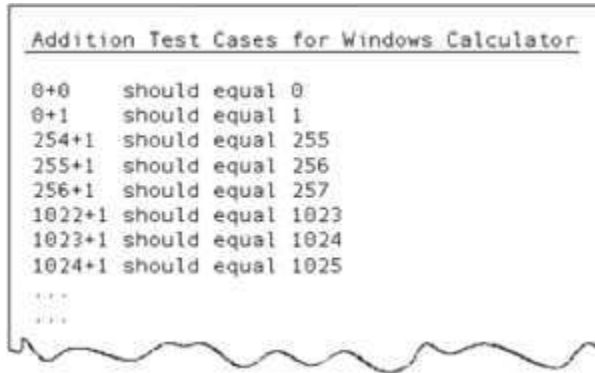
TESTING THE SOFTWARE WITH BLINDERS ON

1.4 Dynamic Black Box Testing

Testing software blindfolded.

- Testing without having insight into details of code
- Dynamic because code is running
- Enter input, receive output and check results

- Another name for **Dynamic testing is behavioral testing**
- Requires production specification or requirement documentation
- Define test cases. Test cases are specific input that is tried
- Test cases are the specific inputs that you'll try and the procedures that you'll follow when you test the software



0+0	should equal 0
0+1	should equal 1
254+1	should equal 255
255+1	should equal 256
256+1	should equal 257
1022+1	should equal 1023
1023+1	should equal 1024
1024+1	should equal 1025

Figure 2.3 Test cases show the different inputs and the steps to test a program.

If test case is improved it result is,

- Too much testing
- Too little testing
- Wrong testing

If no specification is defined, then

- Treat software as specification.
- Take notes on what it does.
- Test with dynamic black box testing.
- Not effective as if with specification.

1.5.Test-to-pass and Test-to-fail

There are two fundamental approaches to testing software: test-to-pass and test-to-fail. When test-to-pass is executed it really assures only that the software minimally works. So it is not pushed to its capabilities. It is not seen what is done to break it. It is treated with kid gloves, applying the simplest and most straightforward test cases.

Two Approaches to testing

- Test-to-pass: apply simple and straight forward test cases.
- Test-to-fail : intend to find bugs by any means

Example: Testing newly developed car

- Test-to-fail: run at full speed.
- Test-to-pass : low speed, normal driving condition

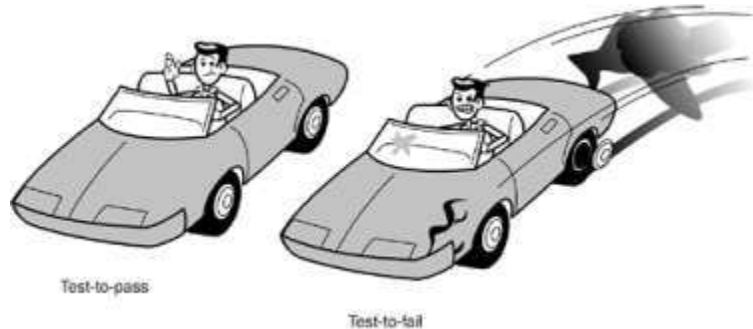


Figure 2.4 Use test-to-pass to reveal bugs before you test-to-fail.

While designing test cases

- Run test-to-pass cases first. If assures that it perform its normal work then go for test-to-fail.
- Test-to-fail (error forcing)

1.6. Equivalence partitioning (or) Equivalence classing

- It is means by which test cases are selected.
- Process of reducing huge set of possible test cases into smaller ones. But equally effective.

Example : - calculator

- Not possible to check all cases of adding 2 numbers together.
- Check 1+1, 1+2, 1+3, safely assure if 1+5, 1+6 also works correct.

Example :

1. 1+9999999999999999 looks different and so may have a bug in it.
2. We provide five options to copy and paste. But all options perform same operation
 - a) Click copy
 - b) type c or C if menu displayed
 - c) ctrl + C or ctrl + shift + C
 - d) Click command to menu
 - e) press Ctrl + C



Figure 2.5 Multiple ways to invoke the copy function with same result.

3. Giving name in **Save As** dialog box - A name must be checked for a valid character, invalid character, valid length, name too short and name too long.



Figure 2.6 File Name text box in the Save As dialog box illustrates several equivalence partition possibilities.

A Windows filename can contain any characters except \ / : * ? " < > and |. Filenames can have from 1 to 255 characters. If test cases are created for filenames, have equivalence partitions for valid characters, invalid characters, valid length names, names that are too short, and names that are too long.

Goals of Equivalence partitioning

- The aim of equivalence partitioning should not be to reduce number of test cases
- This process may lead to bugs.

- If the person is new to testing then get classes from an experienced person.

1.7. Data Testing

- Divide software into data and program
- Data – input, output, printout, mouse clicks etc.,
- Program – flow, transitions, logic, computation.

Examples of data

- Words typed in word processor.
- Numbers typed in spreadsheet
- Number of shots in your game
- Picture printed by your software
- Backup files stored on floppy disk
- Data sent to modem over phone lines.

Tester should reduce test cases by Equivalence partitioning based on few concepts. They are boundary conditions, sub-boundary conditions, nulls and bad data.

1. Boundary conditions

- If it is possible to walk along the edge of a cliff, then it also possible to walk on the middle.
- If software operates on edge of its capabilities, almost operates under normal condition.

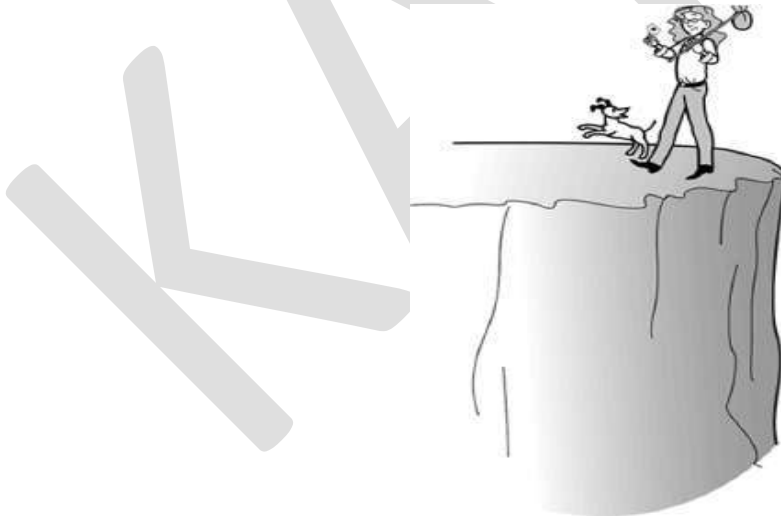


Figure 2.7 Software boundary is much like the edge of a cliff.

Basic program

- 1) Rem create a 10 element integer array
- 2) Rem initialize each element to -1
- 3) Dim data(10) as integer
- 4) Dim i as integer
- 5) For i = 1 to 10
- 6) Data(i) = -1
- 7) Next i
- 8) End

- This program actually creates a data array of 11 elements from data (0) to data (10).
- The program loops from 1 to 10 and initializes those values of the array to 1, but since the first element of our array is data (0), it doesn't get initialized.
- When the program completes, the array values look like this:

data(0) = 0	data(6) = 1
data(1) = 1	data(7) = 1
data(2) = 1	data(8) = 1
data(3) = 1	data(9) = 1
data(4) = 1	data(10) = 1
data(5) = 1	

- The data (0)'s value is 0, not 1.
- If the same programmer later forgot about, or a different programmer wasn't aware of how this data array was initialized, he might use the first element of the array, data (0), thinking it was set to 1.
- Problems such as this are very common and, in large complex software, can result in very nasty bugs.

Types of Boundary Conditions

Boundary conditions are situations at edge of planned operational limits of free software.

When a tester is presented with a software test problem that involves identifying boundaries, he must look for the following types:

Numeric	Speed
---------	-------

Numeric	Speed
Character	Location
Position	Size
Quantity	

And, the following are the characteristics of those types:

First/Last	Min/Max
Start/Finish	Over/Under
Empty/Full	Shortest/Longest
Slowest/Fastest	Soonest/Latest
Largest/Smallest	Highest/Lowest
Next-To/Farthest-From	

Testing the Boundary Edges

- Create 2 Equivalence partitions
- First partition should be such that it is the last value of a data and 2 points to be chosen inside boundary.
- Second partition should be chosen as the data that can cause error and 2 invalid points outside boundary

Testing outside boundary

- First -1 / last + 1
- Start -1 / finish + 1
- Less than empty / more than full
- Even slower / even faster
- Largest+1 / smallest -1

Testing

- Text allowing 1-255 character. Test by entering 1 and 255
- Flight simulator – try at ground level and maximum height allowed
- If software allows 9 digit zip code enter and test 000000000 and 999999999

Sub Boundary Condition

- Not important for end user but must be checked.
- It is also called internal boundary conditions.

Example: ASCII codes; powers-of-two

- These conditions are discussed with programmers and checked.
- It checks for default, empty, blank, null, zero and more.
- It also checks data that are invalid, wrong, incorrect and garbage data
- It includes testing the logic flow of software

KAHE

1.8. State testing

- State testing is performed for verification of program logic
- Software state – It is the condition that the software is currently in.
Example: paint program using menus or clicking button changes its state.
 - In paint program a blank page may be called as a static state
 - Selecting a pencil tool and drawing changes its state.

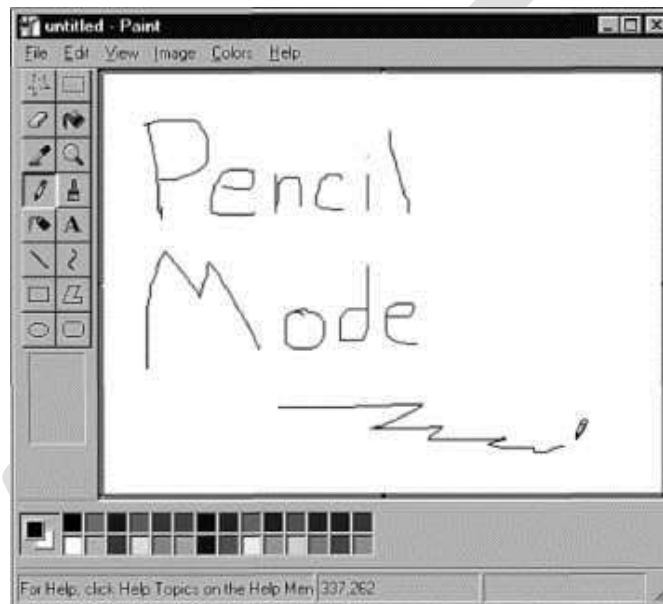


Figure 2.8 Windows Paint program in the pencil drawing state.



Figure 2.9 Windows Paint program in the airbrushing state.

KAHE

A software tester must test a program's states and the transitions between them.

- 1) Testing software logic flow:
 - If the program to be tested is difficult it is impossible to traverse through all possible paths.
 - In such cases apply equivalence partitioning to select state and paths.
- 2) Creating a transition map
 - If transition map is provided in production specification then test it according to the specification
 - If specification is not provided prepare new one and start testing

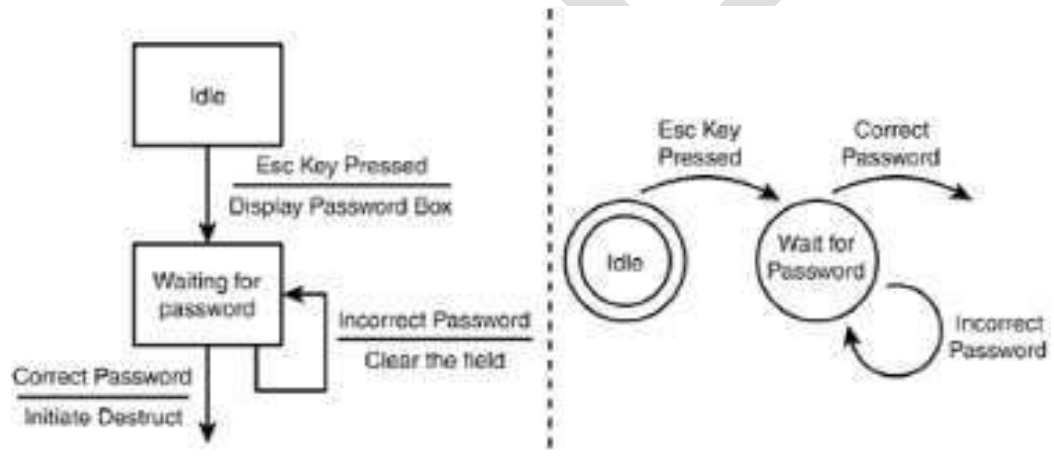


Figure 2.10 State transition diagrams can be drawn by using different techniques

Items in State Transition Map

The following may be the components of a state transition map

- State of software that is unique.
- Input that takes software to another state.
- Set conditions and produced output when state is entered or exited.
- Create static transition map from user's view.

3) Reducing no of states and transitions to test

Following are the five ways to set reasonable amount of states using equivalence partitioning.

- 1) Visit each state at least once.
- 2) Test state-to-state transition that looks popular
- 3) Test least common paths between states.
- 4) Test all error state and returning from error states
- 5) Test random state transitions.

4) What to specifically test

- Identify specific states and state transitions.
- Define test cases

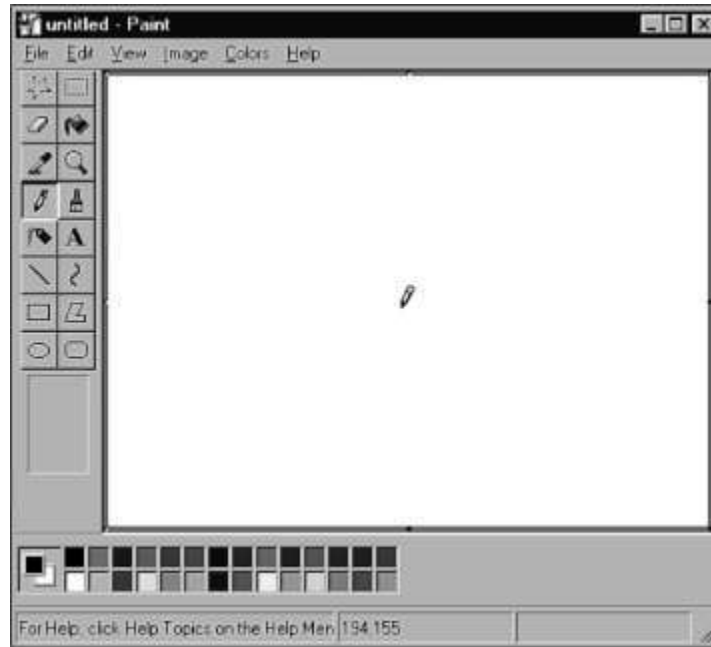


Figure 2.11 Windows Paint opening screen in the startup state

Startup state of paint

- Window size set to previous opened window
- Drawing area is black
- Toolbox, color box and status bar is displayed
- Pencil tool is selected
- Default color black foreground and white background.

Dirty Document Flag:

- It is an invisible state but may be important
- Example: if word document is opened dirty document flag is in “**clean state**”.
- If anything is typed the state changes to **dirty**.
- For every event this bit changes

5) Testing states to fail:

Some of the conditions that lead to test-to-fail is race condition, repetition stress and load.

1. Race condition and bad timing

- Handle interrupt at any time
- Run concurrently with everything else on the system
- Share resource like memory, disk, communication, hardware etc.,

Bad timing - two or more events line up and confess software that didn't expect to be interrupted.

Race condition

- Saving and loading same document at same time by two programs.
- Sharing same printer.
- Pressing keys or mouse click when software is loading.
- Shutting down or sharing up two or more instance of software at same time.
- Different program accessing common database.

b) Repetition, Stress and load

- **Repetition:** Doing same operation over and over
System behave erratically over time
- **Stress:** Running software under less than ideal condition.
Example: slow memory, low disturbance, slow CPU etc.
- **Load:** Feed software with largest possible data files.

2 considerations:

- Testers may say "User will not use the software like this". Convince them to test and find bugs
- It is not possible to check these conditions manually. So find automation software for testing.

1.9. Other black box testing techniques

1. Behave like a dumb user
 - Act like a new in-experienced user.
 - Throw out any preconceived ideas about software.
2. Look for bugs where you have already found them
 - Once the error is found correct it.
 - Check again with same input and more than that limit.
3. Think like a hacker
4. Follow experience, intuition and hunches
 - Gain experience
 - Learn to test different types and size of products.
 - Try different approaches



KARPAGAM ACADEMY OF HIGHER EDUCATION

DEPARTMENT OF COMPUTER SCIENCE

III B.Sc IT (Batch 2017-2020)

SOFTWARE TESTING (17ITU501B)

PART - A OBJECTIVE TYPE/MULTIPLE CHOICE QUESTIONS

ONLINE EXAMINATIONS

ONE MARKS QUESTIONS

UNIT 1

S.NO	Question	Option 1	Option 2	Option 3	Option 4	Answer
1	Black box testing is sometime referred as _____.	Glass	Functional	Structural	All the above	Functional
2	The term ANSI is referred _____	Glass	Functional	Structural	American National Standards	Glass
3	White box testing is sometime called as _____.	American Networki ng	American National Standard	American National Standardize	To Allow	National Standards
4	Usage of equal partitioning is _____ test cases	To Reduce	To Increase	To Avoid	Equivalenc e Directory	To Reduce
5	Other name for sub boundary condition _____	Internal Boundary Condition	External Boundary Condition	Incoming Boundary Value	Quality Condition	Boundary Condition
6	State testing is performed for _____	Verificati on Of Programm	Verificatio n Of Programmi	Transmissi on Of Programmi	Both A & B	Of Programming
7	Other name for equal partitioning _____	Equivalen ce Classes	Equivalenc e Objects	Equivalenc e Methods	Code Analysis	Equivalence Classes
8	Multiple set of condition in the testing called as	Library	Procedure	Race Condition	Both A & B	Race Condition

9	Static white box testing is the process of examining _____	Design	Code	Logic	International Electro Technical	Both A & B
10	Review in software design , architecture or code for bugs is called _____	Logic Analysis	Structural Analysis	Design Analysis	International Organizing	Structural Analysis
11	Structural analysis can be tested by _____	Programmers	Testers	End User	None Of The Above	Both A & B
12	IEC stands for _____	International Engineering	International Engineering	International Electro Technical	American Computer Module	Engineering Consortium
13	ISO stands for _____	International Standard	International Software Organization	International Organization	Institution For Electrical	Organization For Standards
14	NCITS stands for _____	National Cooperation For	National Committee For	National Conference For	Test Drivers	Committee For
15	ACM stands for _____	Association For Computing	American Computing Machinery	Association For Coding Machinery	Test Drivers	For Computing
16	IEEE stands for _____	Institute For	For Electrical	For Electrical	Instruction	Institute For Electrical And Electronic
17	Bottom – up right own modules are called _____	Drivers	Dataflow	Big-Bang	Instruction	Test Drivers
18	Top-down sometime called as _____	Drivers	Dataflow	Big-Bang	Instruction	Big-Bang
19	A process block is a sequence of program statements uninterrupted by	Decisions	Process Block	Case Statement	None Of The Above	Decisions
20	A _____ is a program point at which the control flow can diverge	Decisions	Process Block	Case Statement	None Of The Above	Decisions

21	A _____ is a multi-way branch or decisions	Decisions	Process Block	Case Statement	None Of The Above	Case Statement
22	_____ Execute all possible control flow paths through the program	Path Testing	Statement Testing	Branch Testing	Self Blindness	Path Testing
23	_____ Execute all statement in the program at least once under some test	Path Testing	Statement Testing	Branch Testing	Path	Statement Testing
24	_____ Execute enough tests to assure that every branch alternative has been	Path Testing	Statement Testing	Branch Testing	Path	Branch Testing
25	Predicates of the form A OR B, A AND B and more complicated Boolean expressions	Compound Predicates	Associated Predicates	Assignment Blindness	Case Statements	Compound Predicates
26	The input for a particular test is mapped as a one dimensional array called as	Dynamic Vector	Input Vector	Predicate	No.Of Path	Input Vector
27	The logical function evaluated at a decision is called _____	Dynamic Vector	Input Vector	Predicate	Transmission Of Programmi	Predicate
28	A _____ is a point in the program where the control flow can merge	Process Block	Decisions	Junctions	Structural d)All the above	Junctions
29	The length of path is measured by _____ In it	No.Of Links	No.Of Nodes	No.Of Branches	Equivalence Partitioning	No.Of Links
30	_____ occurs when the buggy predicate is a multiple of the correct	Testing Blindness	Assignment Blindness	Equality Blindness	Self Blindness	Self Blindness
31	Testing helps to	Fix defect	Improve quality	Measure quality	All of the above.	Measure quality
32	Bug is same name of	Error	Incident	Mistake	Defect	Defect

33	Which of the following is largest bug producer?	Code	Design	Specification	other	Specification
34	In software development life cycle , who is the best person to catch a defect?	Software Tester	Customers	Designer	Developer	Business Analyst
35	Defects are less costly if detected in which of the following phases	Coding	Design	Requirements Gathering	Implementation	Implementation
36	User Acceptance testing is	User Acceptance testing	Black box testing	Gray box testing	unit testing	Black box testing
37	. Error guessing is a	Test verification techniques	Test execution techniques	Test control management techniques	Test data management technique	Test data management technique
38	Which of the following term describes testing?	Finding broken code	Evaluating deliverable to find	A stage of all projects	None of the mentioned	deliverable to find errors
39	What are the various Testing Levels?	Unit Testing	System Testing	Integration Testing	All of the mentioned	All of the mentioned
40	Boundary value analysis belong to?	Boundary value analysis	Black Box Testing	White Box & Black Box	Gray-box testing	Black Box Testing
 is black-box testing method that divides the input domain of a program into classes of data from which test cases can be	Condition testing	Graph-based testing	Equivalence partitioning	loop testing	Equivalence partitioning
41 is the first step in black-box testing in order to understand the objects that are modeled in software and the relationships that	Condition testing	Graph-based testing	Comparison testing	loop testing	Graph-based testing

42	The independent versions from the basis of a black-box testing technique are called	Condition testing	Graph-based testing	Comparison testing	loop testing	Comparison testing
43 tests are designed to validate functional requirements without regard to the internal working of program	White-box test	Control structure test	Black-box test	Gray-box test	Black-box test
44	Which of the following is black box testing	Basic path testing	Boundary value analysis	Code path analysis	None of the mentioned	Boundary value analysis

UNIT-II SYLLABUS

Examining the code

Static White-Box testing- Formal reviews – Coding Standards and Guidelines- Generic Code Review Checklist. Testing the software with X-Ray glasses: Dynamic White-Box testing- Dynamic White-Box testing versus Debugging-Testing the Pieces- Data Coverage- Code Coverage.

Flowgraphs and Path Testing

Path-testing Basics – Predicates, Path Predicates and Achievable Paths-Path sensitizing-Path Instrumentation-Implementation and Application of Path Testing

EXAMINING THE CODE

1.10. Static white box testing

It is the process of examining design and code. It includes

- Reviewing software design, architecture or code for bugs with execution.
- Also called structured analysis
- This method finds bugs early during the development process
- It finds bugs that are difficult to uncover
- Highly cost effective
- Many companies treat this time consuming
- But nowadays people have identified its importance

Person who perform this testing

1. Programmers
2. Testers

1.11 Formal review

- It is a meeting between two programmers
- It is a rigorous method of inspection of software design and code.

4 elements:

1. Identify problem - find wrong and missing items. The criticism is only to the product. It should not extend to the programmer
2. Follow rules - amount of code to be reviewed, time spent, what can be commented on etc. are provided in the rules
3. Prepare for review – the reviewers must know what is their duty, role, responsibility during the review and fulfill them.
4. Write a report: Finally after the review a written report summarizing the result of review must be prepared.

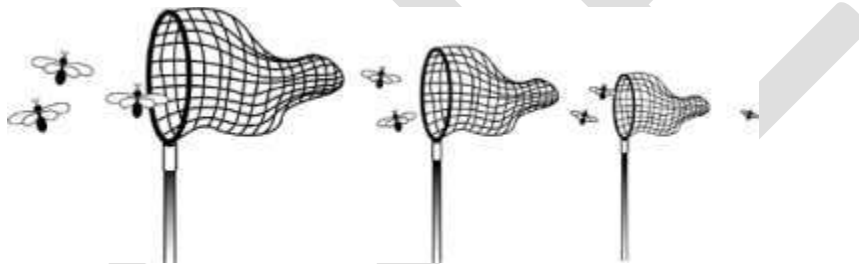


Figure 2.12 Formal reviews are the first nets used in catching bugs

Use of formal review

- It increases communication between testers
- Improves quality of the software product
- Team camaraderie
- It provides a solution even for tough problem.

1. Peer review or buddy review

- It is done by designer, programmer and tester together.
- They review the code and look for problems
- Since it is informal method of testing, the testers may not follow the four elements of testing

2. Walkthroughs

- Programmer who wrote the code formally presents the application to a small group of reviewers

- One senior programmer must be a reviewer
- The reviewer can write comments and questions
- There will be large number of people. So this method looks much formal
- Presenter writes report on how bugs were found and how to solve it

3. Inspections

- It is a highly structured method of testing
- The participants here needs training
- The presenter is not a programmer.
- Some other person learn and explain the code to others
- Other participants are called inspectors
- Using this method identify different bugs
- The review process is done backwards
- After the testing process is done, prepare a written report.
- This report identifies rework
- A re-inspection is done to locate remaining bugs

1.12 Coding standards and guidelines

Standards are specifications that are established, fixed and **have-to-follow-them** rules. It also specifies the rules that must be followed and that must not be followed.

Guidelines may be a bit loose rule. There are 3 rules to adhere to standards and guidelines

1. Reliability – code is reliable if standard is followed
2. Readability/Maintainability – Easy to read, understand and maintain
3. Portability – code can be moved to any platform

Example of programming standards and guidelines

These standards and guidelines specified below are defined for use of ‘goto’, ‘while’ and ‘if-else’ in C

The 4 parts to be mentioned for standard and guideline

1. Title describes what topic the standard covers.

2. Standard (or guideline) describes the standard or guideline explaining exactly what's allowed and not allowed.
3. Justification gives the reasoning behind the standard so that the programmer understands why it's good programming practice.
4. Example shows simple programming samples of how to use the standard. This isn't always necessary.

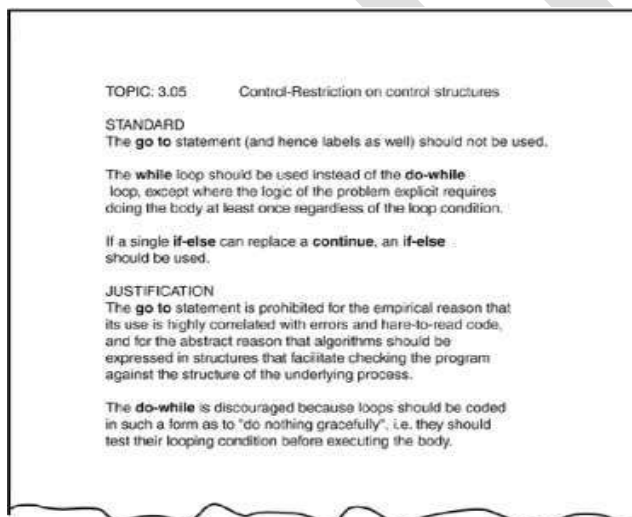


Figure 2.13 Sample coding standard

Example for guideline

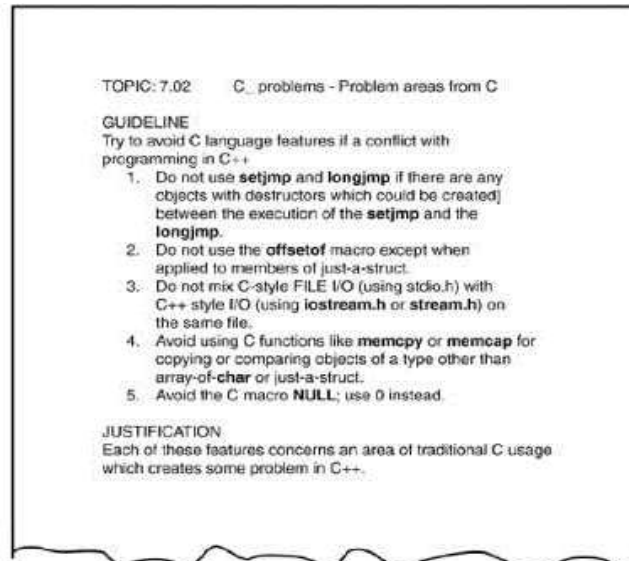


Figure 2.14. An example of a programming guideline

Standards obtained from

1. ANSI – American National Standards Institute
2. IEC – International Engineering Consortium
3. ISO – International Organization for Standardization
4. NCITS – National Committee for Information Technology Standards

Guidelines obtained from

1. ACM – Association for Computing Machinery
2. IEEE – Institute for Electrical and Electronics Engineering Inc.

1.13. Generic Code Review Checklist

There are many types of errors that must be listed before going into the process of testing. They are

1. Data reference error

- Is uninitialized variable referenced?
- Is array out of bound
- Is there any problem in index references of array?
- Is any variable used instead of constant?
- Is floating point number assigned to integer variable?
- Is memory allocated for pointers?

2. Data declaration error

- Is all variables assigned correct length, type etc?
- Check if variable initialized while declared
- Is there variable name with similar name?
- Are there any unreferenced variable
- Are all variable explicitly declared?

3. Computation error

- Existence of two variables of different data type – integer and float
- Existence of two variable of different length – byte and word
- Variable in assignment smaller than Right hand side of expression
- Overflow, underflow
- Division by zero
- Value of variable outside range. Example percentage value only between 0-100
- Confusion in order of expression if there are multiple operators

4. Comparison error

- Is comparison correct?
- Is there comparison between fractional and floating point?
- Is there any confusion in order of evaluation in Boolean expression?
- Whether operators in Boolean expression are Boolean?

5. Control flow error

- Matching groups, begin-end, do-while
- Whether loop terminates correctly?
- Possibility if a loop never executes
- Switch index exceeds number of existing branches
- Unexpected flow through loop

6. Subroutine parameter error

- Type, size and order of precedence correct or not
- Multiple entry point in subroutine
- Change in order of parameters if send as constant
- Is there any alteration in parameters?
- Whether formal and actual arguments match?
- Whether definition of global variable same everywhere

7. I/O error

- Data format read/printed
- Device not ready
- Device disconnected
- Error handled in expected way
- Check error message for spelling and grammar

8. Other checks

- Will the software work with language other than English?
- Will the software work with other compilers and CPU's?
- Will the software work with different amount of memory, hardware, sound card etc?
- Whether compilation of program produce warning or informational message

TESTING SOFTWARE WITH X-RAY GLASSES

1.14. Dynamic White-Box Testing

It is a structured method of testing. This method use underlying structure of code to design and run the tests.

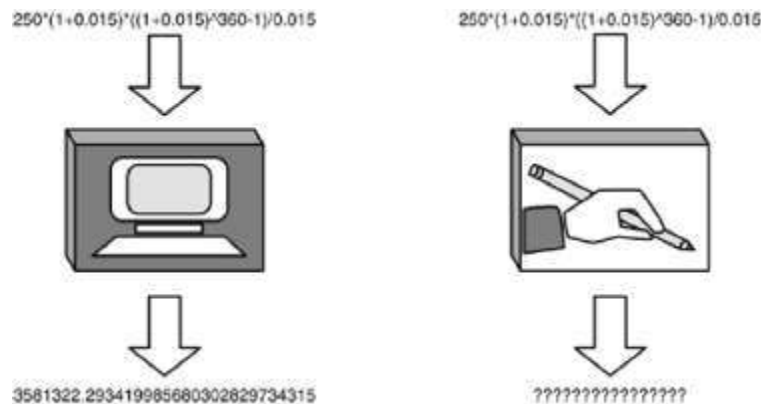


Figure 2.15 A test case containing a computer and other a person with a pencil and paper. The four areas that dynamic white-box testing encompasses are

1. Directly testing low-level functions, procedures, subroutines, or libraries. In Microsoft Windows, these are called Application Programming Interfaces (APIs).
2. Testing the software at the top level, as a completed program, but adjusting your test cases based on what you know about the software's operation.
3. Gaining access to read variables and state information from the software to help you determine whether your tests are doing what you thought. And, being able to force the software to do things that would be difficult if you tested it normally.
4. Measuring how much of the code and specifically what code you "hit" when you run your tests and then adjusting your tests to remove redundant test cases and add missing ones.

1.15. Dynamic white box testing versus debugging

It's important not to confuse dynamic white-box testing with debugging. The two techniques may appear similar because they both involve dealing with software bugs and looking at the code, but they're very different in their goals

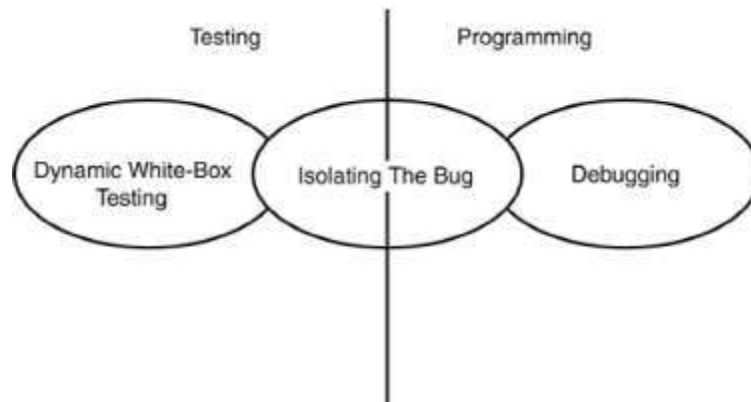


Figure 2.16 Dynamic white-box testing and debugging have different goals but they do overlap in the middle

White box testing – Find bugs to perform testing

Debugging – Programming is done and fix the bugs

Tester – Performs white box testing. He includes information about suspicious code.

Programmer – Debugs and picks process from tester. He determines the reason for bug and fixes it.

1.16. Testing the pieces

Testing cost of software if errors found late in the program

2 reasons for high cost

1. Difficult to find place of error
2. Some bugs hide others

a. Unit and integration testing

Testing at lowest level – unit or module testing

Testing after integration modules – Integration testing

Testing entire or major portion of software – System testing

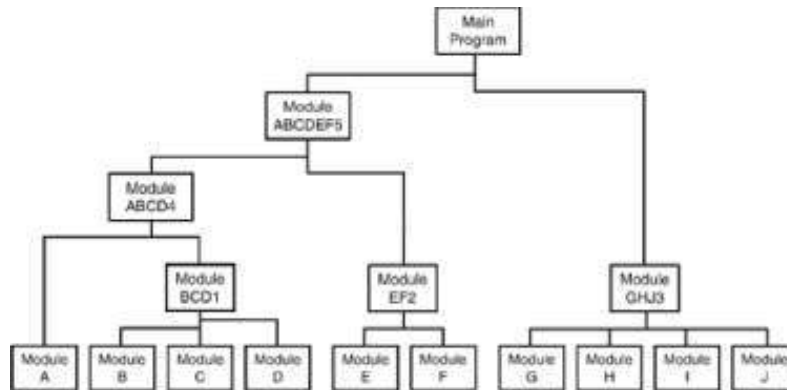


Figure 2.17 Individual pieces of code are built up and tested separately, and then integrated and tested again

b. Approaches to this incremental testing

i) Bottom-up

Write own modules called test drivers

These drivers send test-case data to testing modules

Read results

Verify whether they are correct or not

ii) Top-down

Top-down testing may sound like big-bang testing on a smaller scale.

After all, if the higher-level software is complete

It must be too late to test the lower modules

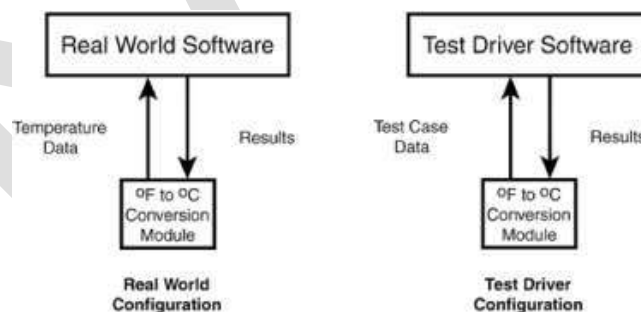


Figure 2.18 A test driver can replace the real software and more efficiently test a low-level module.

1.17. Data Coverage

- A black-box tester is not aware of the 'division by zero' error. He finds the result of execution given the value of n as 0.
 - A white box tester is aware of the 'division by zero' error if the value given for n=0. He checks if there is any way that n can ever become zero
- d. Error forcing
- It is a method used to force the software to test specific values
 - In the compound Interest calculation, even if value of n is not zero, force it to become zero.
 - Check what the software does to handle it.

1.18. Code coverage

This involves the following process

- Enter and Exit every module.
- Execute every line of code
- Follow every logic and decision path through software

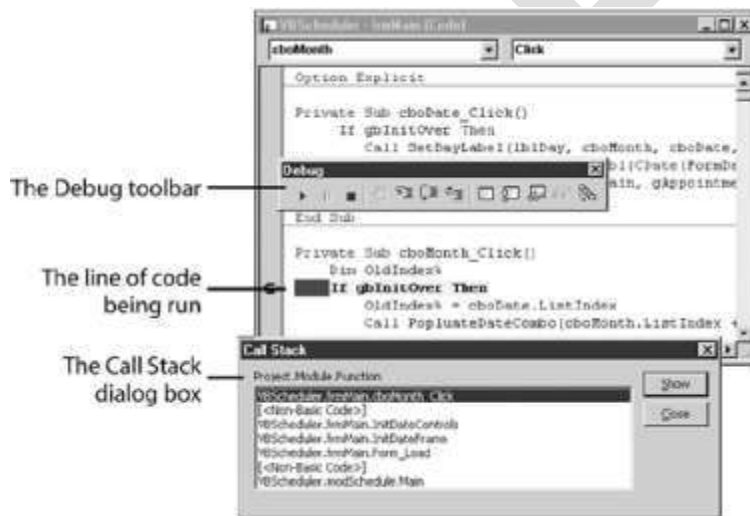


Figure 7.9 Debugger allows to single-step through the software to see lines of code and modules are executed while running test cases.

Process

- Code coverage is a dynamic white box testing method
- This method have full access to code
- This method views all part of the software when test cases are minimum

- The simplest way of code coverage process is using compiler's debugger to view lines of code.
- If the program size is small use the debugger to test
- If the program is large, the tester must use some specialized tool for code coverage analysis
- The code coverage analyzer runs transparently in the background execution of the software. Each time a line of code is executed, the analyzer records the information.
- The results obtained from the analyzer are
 - Which part of software test cases don't cover? If so, write additional test cases
 - Which test cases are redundant? If it exists remove it.
 - What new cases are to be created for better coverage? Find those cases and add to the existing test cases.

Types of Code coverage

a. Program statements and line coverage

- Execute every statement of program atleast once
- But it cannot be assured that all paths of the code are covered fully

b. Branch coverage

- Covering all the paths in the software is called path testing. A form of path testing is branch coverage testing
- It tests all statements and all branches

c. Condition coverage

- It takes extra conditions on branch statements into account
- Example
 - If Date=01-01-2000 and Time=00.00.00, then test cases should be

▪ 01-01-1999	11.11.11
▪ 01-01-1999	00.00.00
▪ 01-01-2000	11.11.11
▪ 01-01-2000	00.00.00

- Check for condition coverage and achieve branch coverage and automatically statement coverage is achieved

FLOWGRAPHS AND PATH TESTING

2.1 Path Testing Basics

2.1.1. Motivations and Assumptions

- **Path Testing**

- Path Testing is the name given to a family of test techniques based on judiciously selecting a set of test paths through the program.
- If the set of paths are properly chosen then we have achieved some measure of test thoroughness. For example, pick enough paths to assure that every source statement has been executed at least once.
- Path testing techniques are the oldest of all structural test techniques.
- Path testing is most applicable to new software for unit testing. It is a structural technique.
- It requires complete knowledge of the program's structure.
- It is most often used by programmers to unit test their own code.
- The effectiveness of path testing rapidly deteriorates as the size of the software aggregate under test increases.

- **The Bug Assumption**

- The bug assumption for the path testing strategies is that something has gone wrong with the software that makes it take a different path than intended.
- As an example "GOTO X" where "GOTO Y" had been intended.
- Structured programming languages prevent many of the bugs targeted by path testing: as a consequence the effectiveness for path testing for these languages is reduced and for old code in COBOL, ALP, FORTRAN and Basic, the path testing is indispensable.

2.1.2. Control Flow Graphs

- The control flow graph is a graphical representation of a program's control structure. It uses the elements named process blocks, decisions, and junctions.
- The flow graph is similar to the earlier flowchart, with which it is not to be confused.
- **Flow Graph Elements:** A flow graph contains four different types of elements. (1) Process Block (2) Decisions (3) Junctions (4) Case Statements

1. **Process Block:**

- A process block is a sequence of program statements uninterrupted by either decisions or junctions.

- It is a sequence of statements such that if any one of statement of the block is executed, then all statement thereof is executed.
- Formally, a process block is a piece of straight line code of one statement or hundreds of statements.
- A process has one entry and one exit. It can consists of a single statement or instruction, a sequence of statements or instructions, a single entry/exit subroutine, a macro or function call, or a sequence of these.

2. Decisions

- A decision is a program point at which the control flow can diverge.
- Machine language conditional branch and conditional skip instructions are examples of decisions.
- Most of the decisions are two-way but some are three way branches in control flow.

3. Case Statements

- A case statement is a multi-way branch or decisions.
- Examples of case statement are a jump table in assembly language, and the PASCAL case statement.
- From the point of view of test design, there are no differences between Decisions and Case Statements.

4. Junctions

- A junction is a point in the program where the control flow can merge.
- Examples of junctions are: the target of a jump or skip instruction in ALP, a label that is a target of GOTO.

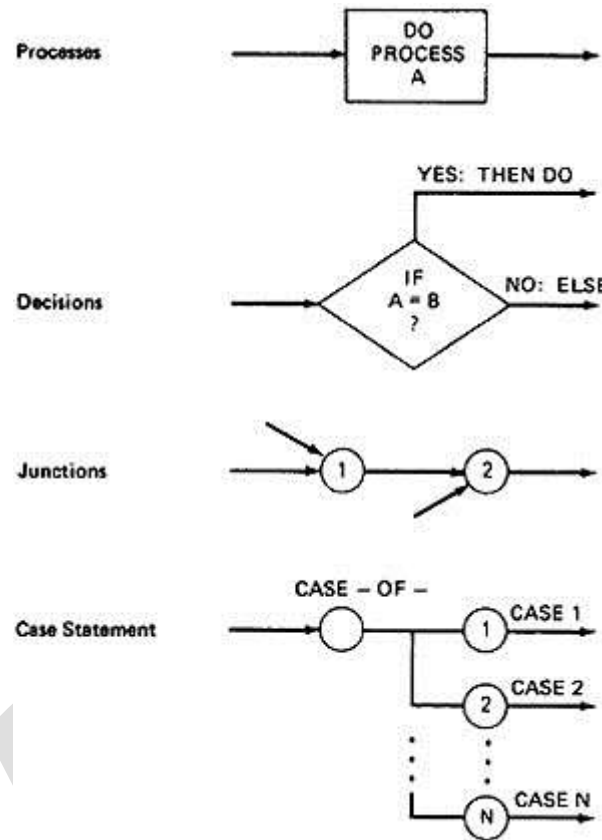


Figure 2.1: Flowgraph Elements

2.1.3 Path Testing

Paths, Nodes and Links

- A path through a program is a sequence of instructions or statements that starts at an entry, junction or decision and ends at another or possibly the same junction, decision or exit.
- Path goes through several segments. The smallest segment is a link that is 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 path is measured by the no. of links in it.
- The name of the path is the name of the nodes along the path
- The path has a loop in it if any node name is repeated.

Multi-Entry/Multi-Exit Routines

The trouble with multi- entry and multi-exit routines is that it can be very

difficult to determine what the interprocess control flow is. The use of multi-entry and multi-exit routines increases the number of entries and exits and therefore the number of interfaces.

Fundamental Path Selection Criteria

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.

Path –Testing Criteria

We have, therefore, explored three different testing criteria or strategies out of a potentially infinite family of strategies

- 1.Path Testing – Execute all possible control flow paths through the program.
- 2.Statement Testing-Execute all statement in the program at least once under some test.
- 3.Branch Testing –Execute enough tests to assure that every branch alternative has been exercised at least once under some test.

2.1.4. Loops

The Kinds of Loops - There are only three kinds of loops: nested, concatenated and horrible.

Cases for a Single Loop

A single loop can be covered with two cases: looping and not looping.

Case 1-Single Loop, Zero Minimum, N Maximum, No Excluded Values

1. Try bypassing the loop.
2. Could the loop-control variable be negative? Could it appear to specify a negative number of iterations? What happens to such a value?
3. One pass through the loop.
4. Two passes through the loop for reason discussed below.
5. A typical number of iterations, unless covered by a previous test.
6. One less than the maximum number of iterations.
7. The maximum number of iterations.
8. Attempt one more than the maximum number of iterations.

Case 2-Single Loop, Nonzero Minimum, No Exclude Values

1. Try one less than the expected minimum. What happens if the loop control variable's value is less than the minimum? What prevents the value from being less than the minimum?
2. The minimum number of iterations.
3. One more than the minimum number of iterations.
4. Once, unless covered by a previous test.
5. Twice, unless covered by a previous test.

6. A typical value.
7. One less than the maximum value.
8. The maximum number of iterations.
9. Attempt one more than the maximum number of iterations.

Case 3-Single Loops with Excluded Values

Treat single loops with excluded values as two sets of tests consisting of loops without excluded values, such as Cases 1to2.

Nested Loops

1. Start at the innermost loop. 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.
3. If done with outer most loop go to step 5 else move out one loop and set it up as in step 2
4. Continue outward in this manner until all loops have been covered.
5. Do the five cases for all loops in the nest simultaneously

Concatenated Loops

This loop falls between single and nested loops with respect to test cases. Two loops are concatenated if it is possible to reach one after exiting the other while still on a path from entrance to exit. If the loops cannot be on the same path, then they are not concatenated and can be treated as individual loops.

Horrible Loops

It is use of code that jumps into and out of loops, intersecting loops, hidden loops and cross-connected loops make iteration-value selection for test cases which is an ugly task that should be avoided

2.2 Predicates, Path Predicates and Achievable Paths

2.2.1 Predicate

The logical function evaluated at a decision is called Predicate. The direction taken at a decision depends on the value of decision variable. Some examples are: $A > 0$, $x + y \geq 90$

Path Predicate

A predicate associated with a path is called a Path Predicate. For example, "x is greater than zero", " $x + y \geq 90$ ", "w is either negative or equal to 10 is true" is a sequence of predicates whose truth values will cause the routine to take a specific path.

Multiway Branches

- The path taken through a multiway branch such as a computed GOTO's, case statement, or jump tables cannot be directly expressed in TRUE/FALSE terms.
- Although, it is possible to describe such alternatives by using multi valued logic, an expedient (practical approach) is to express multiway branches as an equivalent set of if..then..else statements.
- For example a three way case statement can be written as: If case=1 DO A1 ELSE (IF Case=2 DO A2 ELSE DO A3 ENDIF)ENDIF.

Inputs

- In testing, the word input is not restricted to direct inputs, such as variables in a subroutine call, but includes all data objects referenced by the routine whose values are fixed prior to entering it.
- For example, inputs in a calling sequence, objects in a data structure, values left in registers, or any combination of object types.
- The input for a particular test is mapped as a one dimensional array called as an Input Vector.

2.2.2 Predicate Expressions

Predicate Interpretation

- The simplest predicate depends only on input variables.
- For example if x_1, x_2 are inputs, the predicate might be $x_1 + x_2 \geq 7$, given the values of x_1 and x_2 the direction taken through the decision is based on the predicate is determined at input time and does not depend on processing.
- Another example, assume a predicate $x_1 + y \geq 0$ that along a path prior to reaching this predicate we had the assignment statement $y = x_2 + 7$. although our predicate depends on processing, we can substitute the symbolic expression for y to obtain an equivalent predicate $x_1 + x_2 + 7 \geq 0$.
- The act of symbolic substitution of operations along the path in order to express the predicate solely in terms of the input vector is called **predicate interpretation**.

- Some times the interpretation may depend on 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 on the path we took through the first multi way branch. It yields for the three cases respectively, if $Y+7>0$, $Y-7>0$, $Y>0$.

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

Independence Of Variables And Predicates

- The path predicates take on truth values based on the values of input variables, either directly or indirectly.
- If a variable's value does not change as a result of processing, that variable is independent of the processing.
- If the variable's value can change as a result of the processing, the variable is process dependent.
- A predicate whose truth value can change as a result of the processing is said to be **process dependent** and one whose truth value does not change as a result of the processing is **process independent**.
- Process dependence of a predicate does not always follow from dependence of the input variables on which that predicate is based.

Correlation Of Variables And Predicates

- Two variables are correlated if every combination of their values cannot be independently specified.
- Variables whose values can be specified independently without restriction are called uncorrelated.
- A pair of predicates whose outcomes depend on one or more variables in common are said to be correlated predicates. For example, the predicate $X=Y$ is followed by another predicate $X+Y=8$. If we select X and Y values to satisfy the first predicate, we might have forced the 2nd predicate's truth value to change.
- Every path through a routine is achievable only if all the predicates in that routine are uncorrelated.

Path Predicate Expressions

- A path predicate expression is a set of Boolean expressions, all of which must be satisfied to achieve the selected path.

Example:

$$\begin{aligned}X1+3X2+17 &\geq 0 \\X3 &= 17 \\X4-X1 &\geq 14X2\end{aligned}$$

Any set of input values that satisfy all of the conditions of the path predicate expression will force the routine to the path.

Some times a predicate can have an OR in it.

Example:

A: $X5 > 0$	E: $X6 < 0$
B: $X1 + 3X2 + 17 \geq 0$	B: $X1 + 3X2 + 17 \geq 0$
C: $X3 = 17$	C: $X3 = 17$
D: $X4 - X1 \geq 14X2$	D: $X4 - X1 \geq 14X2$

Boolean algebra notation to denote the Boolean expression:

$$ABCD + EBCD = (A + E)BCD$$

2.2.3. Predicate Coverage

Compound Predicate: Predicates of the form A OR B, A AND B and more complicated Boolean expressions are called as compound predicates.

Some times even a simple predicate becomes compound after interpretation. Example: the predicate if (x=17) whose opposite branch is if x.NE.17 which is equivalent to $x > 17$. Or. $x < 17$.

Predicate coverage is being the achieving of all possible combinations of truth values corresponding to the selected path have been explored under some test.

As achieving the desired direction at a given decision could still hide bugs in the associated predicates.

2.2.4. Testing Blindness

Testing Blindness is a pathological (harmful) situation in which the desired path is achieved for the wrong reason.

There are three types of Testing Blindness:

Assignment Blindness

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

For Example:

Correct	Buggy
$X = 7$	$X = 7$
.....
if $Y > 0$ then ...	if $X + Y > 0$ then ...

If the test case sets $Y=1$ the desired path is taken in either case, but there is still a bug.

Equality Blindness

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

For Example:

Correct	Buggy
if $Y = 2$ then if $X+Y > 3$ then ...	if $Y = 2$ then if $X > 1$ then ...

The first predicate if $y=2$ forces the rest of the path, so that for any positive value of x . the path taken at the second predicate will be the same for the correct and buggy version.

Self Blindness

Self blindness occurs when the buggy predicate is a multiple of the correct predicate and as a result is indistinguishable along that path.

For Example:

Correct	Buggy
$X = A$ if $X-1 > 0$ then ...	$X = A$ if $X+A-2 > 0$ then ...

The assignment ($x=a$) makes the predicates multiples of each other, so the direction taken is the same for the correct and buggy version.

2.3. Path Sensitizing

2.3.1. Review: Achievable and Unachievable Paths

Select and test enough paths to achieve a satisfactory notion of test completeness such as C1+C2.

Extract the programs control flow graph and select a set of tentative covering paths.

For any path in that set, interpret the predicates along the path as needed to express them in terms of the input vector. In general individual predicates are compound or may become compound as a result of interpretation.

Trace the path through multiplying the individual compound predicates to achieve a Boolean expression such as

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

Multiply out the expression to achieve a sum of products form:

$$ADFGHIJKL+AEFGHIJKL+BCDFGHIJKL+BCEFGHIJKL$$

Each product term denotes a set of inequalities that if solved will yield an input vector that will drive the routine along the designated path.

Solve any one of the inequality sets for the chosen path and you have found a set of input values for the path.

If you can find a 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**.

2.3.2. Heuristic Procedures For Sensitizing Paths

- This is a workable approach, instead of selecting the paths without considering how to sensitize, attempt to choose a covering path set that is easy to sensitize and pick hard to sensitize paths only as you must to achieve coverage.
- Identify all variables that affect the decision.
- Classify the predicates as dependent or independent.
- Start the path selection with uncorrelated, independent predicates.
- If coverage has not been achieved using independent uncorrelated predicates, extend the path set using correlated predicates.

- If coverage has not been achieved extend the cases to those that involve dependent predicates.
- Last, use correlated, dependent predicates.

2.4. Path Instrumentation

- Path instrumentation is what we have to do to confirm that the outcome was achieved by the intended path.
- **Co-incidental Correctness:** The coincidental correctness stands for achieving the desired outcome for wrong reason.

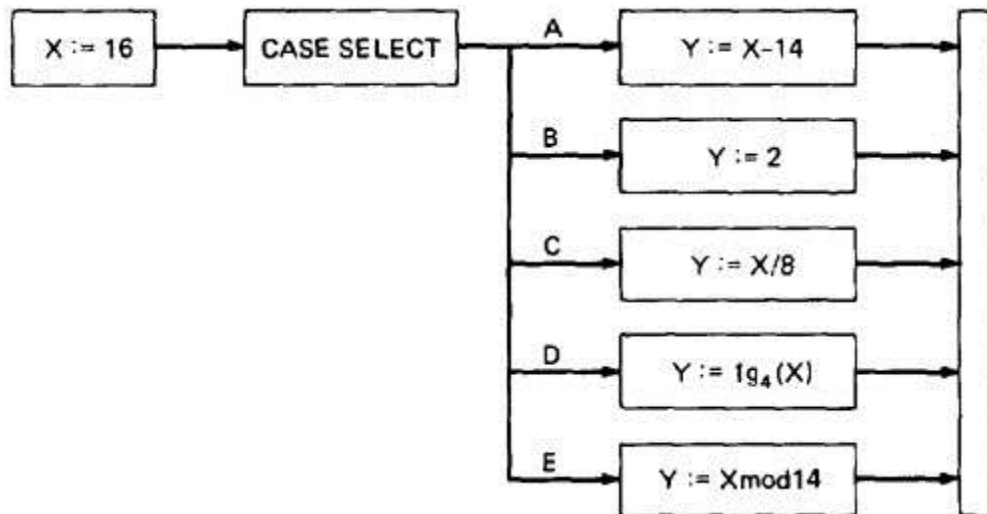


Figure 2.2: Coincidental Correctness

- The above figure is an example of a routine that, for the (unfortunately) chosen input value ($X = 16$), yields the same outcome ($Y = 2$) no matter which case we select. Therefore, the tests chosen this way will not tell us whether we have achieved coverage. For example, the five cases could be totally jumbled and still the outcome would be the same. **Path Instrumentation** is what we have to do to confirm that the outcome was achieved by the intended path.
- The types of instrumentation methods include:

Interpretive Trace Program:

- An interpretive trace program is one that executes every statement in order and records the intermediate values of all calculations, the statement labels traversed etc.
- If we run the tested routine under a trace, then we have all the information we need to confirm the outcome and, furthermore, to confirm that it was achieved by the intended path.
- The trouble with traces is that they give us far more information than we need. In fact, the typical trace program provides so much information that confirming the path from its massive output dump is more work than simulating the computer by hand to confirm the path.

Traversal Marker or Link Marker:

- A simple and effective form of instrumentation is called a traversal marker or link marker.
- Name every link by a lower case letter.
- Instrument the links so that the link's name is recorded when the link is executed.
- The succession of letters produced in going from the routine's entry to its exit should, if there are no bugs, exactly correspond to the path name.

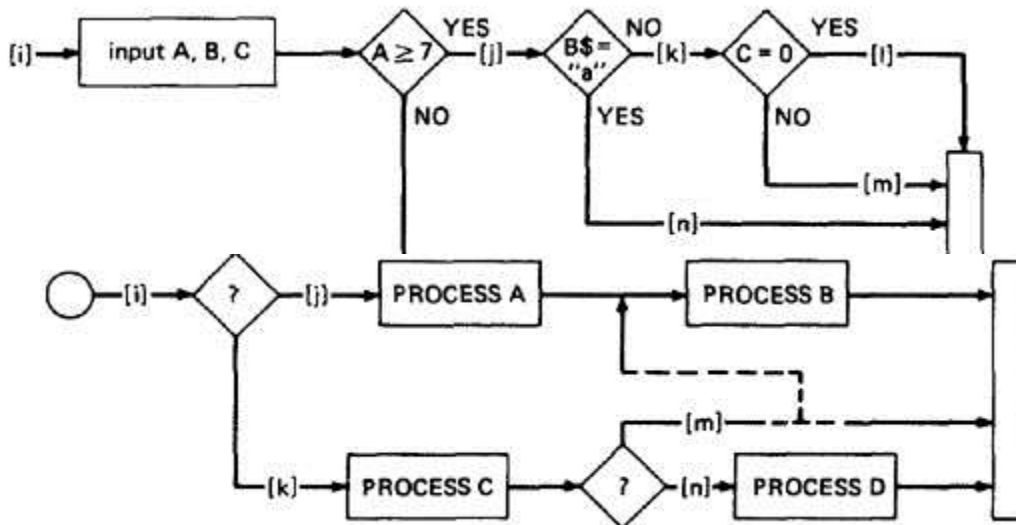
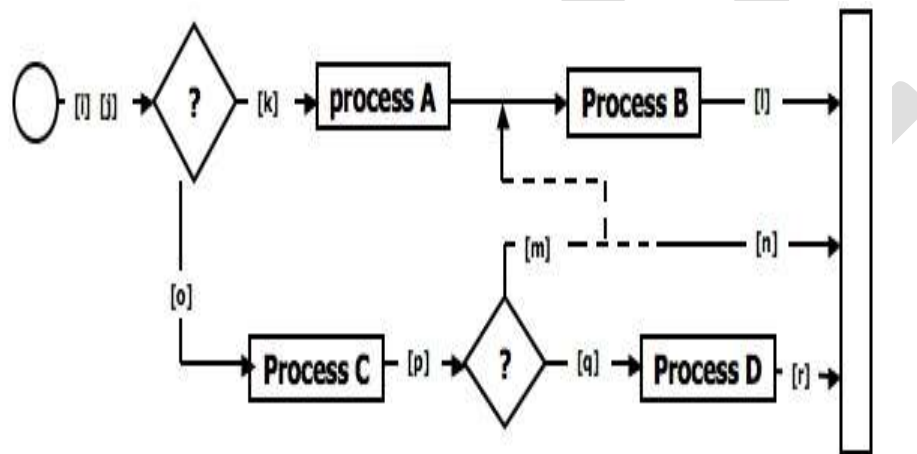


Figure 2.3 Single Link Marker Instrumentation

- Because of a rampaging GOTO in the middle of the m link, we go to process B. If coincidental correctness is against us, the outcomes will be the same and we won't know about the bug.

Two Link Marker Method

- The solution to the problem of single link marker method is to implement two markers per link: one at the beginning of each link and one at the end.



- The two link markers now specify the path name and confirm both the beginning and end of the link.

Figure 2.4: Double Link Marker Instrumentation.

Link Counter

- A less disruptive (and less informative) instrumentation method is based on counters. Instead of a unique link name to be pushed into a string when the link is traversed, we simply increment a link counter. We now confirm that the path length is as expected. The same problem that led us to double link markers also leads us to double link counters.

2.5 Implementation and Application of Path Testing

2.5.1. Integration, Coverage and Path in Called Components

Path testing methods are mainly used in unit testing. However to create an environment in order to provide required inputs and also to receive the outputs from such units, we need to do test harness in order to create environment with required test stubs and test drivers. In order to perform testing in sub routines that need to be integrated, we have to think about paths within the sub routine. Then to achieve test coverage both statement coverage (C1) and branch coverage (C2) are to be done.

The components must be then integrated with its called subroutines by carefully probing the interface issues. Once the integrations have been tested we retest the integrated components.

2.5.2.New Code

Wholly new or substantially modified code should always be subjected to enough path testing to achieve C2.

2.5.3.Maintenance

Path testing is used first on the modified component, as for new software but called and co-requisite components will invariably be real rather than simulated.

2.5.4.Rehosting

Path testing with C1+C2 coverage is a powerful tool for rehosting old software. When used in conjunction with automatic or semi automatic structural test generators, a powerful effective rehosting process can be done.

2.6. Testability tips

1. Keep in mind three numbers: the total number of paths, the total number of achievable Paths and the number of paths required to achieve C2 coverage.
2. Make decision once and only once and stick to them
3. Don't squeeze the code
4. If you can't test it, don't build it
5. If you don't test it rip it out
6. Introduce no extras and unwanted generalizations
7. If it is not possible to sensitize then we don't know what we are doing
8. Easy cover beats elegance every time
9. Covering paths make functional sense
10. Deeply nested and/or horrible loops are not a mark of genius but of a murky mind
11. Flags switches and instruction modification are evil



KARPAGAM ACADEMY OF HIGHER EDUCATION

DEPARTMENT OF COMPUTER SCIENCE

III B.Sc IT (Batch 2017-2020)

SOFTWARE TESTING (17ITU501B)

PART - A OBJECTIVE TYPE/MULTIPLE CHOICE QUESTIONS

ONLINE EXAMINATIONS

ONE MARKS QUESTIONS

UNIT II

S.NO	Question	Option 1	Option 2	Option 3	Option 4	Answer
1	Act of finding set of solutions to the path predicate expression is _____	path segmentation	path sensitization	path predicate	path documentation	path sensitization
2	The combination of sum of products and path predicate expression is _____	Acheivable path	ditrected path	predicate path	branch path	Acheivable path
3	Types of test blindness is _____	4	6	3	7	3
4	_____ is executing all possible control flow path to the program	path testing	statement testing	branch testing	segment testing	path testing
5	_____ is execute all statement atleast once in some testing	path testing	statement testing	branch testing	segment testing	statement testing
6	sequence of process the link and node is _____	branch	segment	path	predicate	path
7	The logical function evaluated at a decision true or false is _____	branch	segment	path	predicate	predicate
8	Two or more predicate combined with AND,OR is _____	compound predicate	branch	segment	path	compound predicate

9	Every path corresponding to a success of true or false for predicate traversed on that path is _____	branch	path predicate	segment	path	path predicate
10	CFG stands for _____	control flow graph	compound flow graph	conditional flow graph	none of these	control flow graph
11	A path testing calculated by _____ complexity from CFG	Decision complexity	cyclomatic complexity	compound complexity	conditional complexity	cyclomatic complexity
12	TFG stands for _____	Transaction flow graph	Timing flow graph	transform flow graph	none of these	Transaction flow graph
13	_____ is a unit of work seen from system, user point of view	Transaction	segmentation	transformation	none of these	Transaction
14	Transaction flow representing the _____	System processing	Time processing	transaction processing	segment processing	system processing
15	_____ types of methods are applied for testing transaction flow	path test	Functional test	branch test	none of these	functional test
16	The types of transaction flow testing is _____	6	2	5	8	8
17	_____ is a every combination of their value cannot be specified independently	segmentation	correlation	transaction processing	none of these	correlation
18	_____ is a single process between two nodes	path	segment	link	node	link
19	_____ is a junction or decision	path	segment	link	node	node
20	_____ is a sequence of links	path	segment	link	node	segment

21	A path consists of any _____	path	segment	link	node	segment
22	_____ is a successive of consecutive links belongs to the same path	path segment	branch segment	segment	none of these	path segment
23	_____ is measured by number of links in the path	height of the path	entry of the path	exit of the path	length of the path	length of the path
24	_____ is set of names in the node along the path	Name of the path	entry of the path	exit of the path	length of the path	Name of the path
25	Path testing path is an _____ path through a processing block	Name of the path	entry of the path	exit of the path	entry to exit	entry to exit
26	Transaction is represented by _____	token	identifier	constant	none of these	token
27	_____ is a pictorial representation of what happen to the token	CFG	CGF	TFG	FFG	TFG
28	_____ is a implicit the design the control structure and the associated database	control flow	segment flow	transaction flow	none of these	transaction flow
29	A _____ containing basic blocks of the program	directed graph	undirected graph	small graph	big graph	directed graph
30	_____ is defined with reference to the control flow graph of the program	structure program	segment program	transaction program	none of these	structure program
31	White Box techniques are also classified as	Design based testing	Structural testing	Error guessing technique	None of the mentioned	Structural testing
32	Which of the following is/are White box technique?	Statement Testing	Decision Testing	Condition Coverage	All of the mentioned	All of the mentioned

33, sometimes called glass-box testing, is a test case design method that uses the control structure of the procedural design to derive test cases.	White-box testing	Control structure testing	Black-box testing	Gray-box testing	White-box testing
34	In, test cases are derived to ensure that all statements in the program have been executed at least once during testing and that all logical conditions have	White-box testing	Control structure testing	Black-box testing	Gray-box testing	White-box testing
35	The testing in which code is checked	Black box testing	White box testing	Red box testing	Green box testing	White-box testing
36	Which of the following is non-functional testing?	Black box testing	Performance testing	Unit testing	None of the mentioned	Performance testing
37	Unit testing is done by	Users	Developers	Customers	None of the mentioned	Developers

COURSE CODE: 17CTU501B

UNIT: III (Transaction-Flow Testing and Data-Flow Testing) BATCH-2017-2020

UNIT-III

SYLLABUS

Transaction-Flow Testing and Data-Flow Testing

Transaction Flows-Transaction Flow Testing Techniques. Data-Flow Testing Basics-Data-Flow Testing Strategies-Application, Tools, Effectiveness

TRANSACTION FLOW TESTING

3.7 Transaction Flows

3.7.1 Definitions

- A transaction is a unit of work seen from a system user's point of view.
- A transaction consists of a sequence of operations, some of which are performed by a system, persons or devices that are outside of the system.
- Transaction begins with Birth-that is they are created as a result of some external act.
- At the conclusion of the transaction's processing, the transaction is no longer in the system.

3.7.2 Example of a transaction

A transaction for an online information retrieval system might consist of the following steps or tasks:

- Accept input (tentative birth)
- Validate input (birth)
- Transmit acknowledgement to requester
- Do input processing
- Search file
- Request directions from user
- Accept input
- Validate input
- Process request
- Update file
- Transmit output
- Record transaction in log and clean up (death)

Transaction Flow Graphs

COURSE CODE: 17CTU501B

UNIT: III (Transaction-Flow Testing and Data-Flow Testing) BATCH-2017-2020

- Transaction flows are introduced as a representation of a system's processing.
- The methods that were applied to control flow graphs are then used for functional testing.
- Transaction flows and transaction flow testing are to the independent system tester what control flows are path testing is to the programmer.
- The transaction flow graph is to create a behavioral model of the program that leads to functional testing.
- The transaction flow graph is a model of the structure of the system's behavior (functionality).
- An example of a Transaction Flow is as follows:

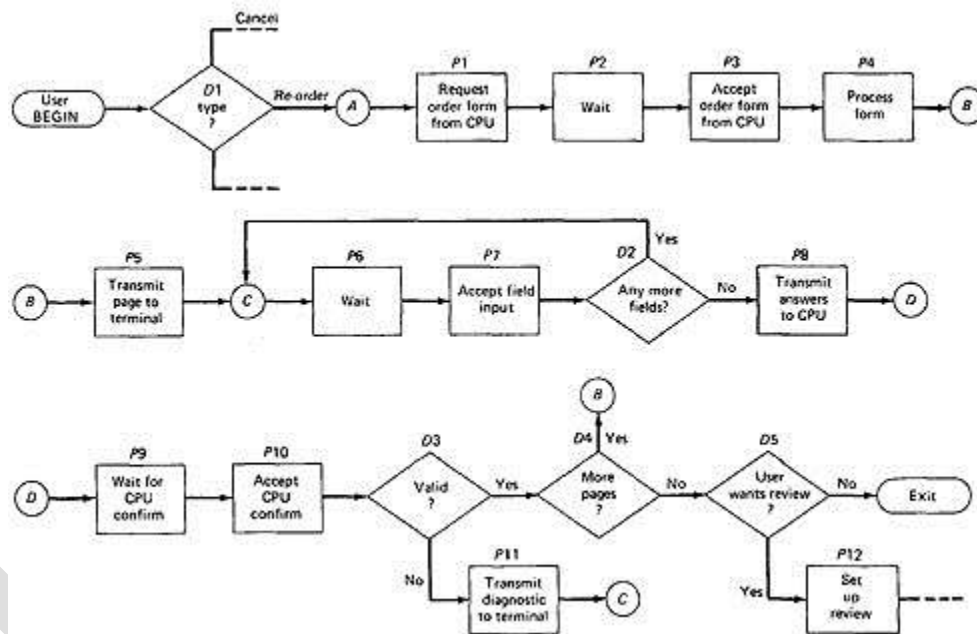


Figure 3.1: An Example of a Transaction Flow

2.7.3 Usage

- Transaction flows are indispensable for specifying requirements of complicated systems, especially online systems.
- A big system such as an air traffic control or airline reservation system, has not hundreds, but thousands of different transaction flows.

COURSE CODE: 17CTU501B

UNIT: III (Transaction-Flow Testing and Data-Flow Testing) BATCH-2017-2020

- The flows are represented by relatively simple flow graphs, many of which have a single straight-through path.
- Loops are infrequent compared to control flow graphs.
- The most common loop is used to request a retry after user input errors. An ATM system, for example, allows the user to try, say three times, and will take the card away the fourth time.

3.7.4 Implementation

- The implementation of transaction flow is usually implicit in the design of the system's control structure and associated database.
- A transaction flow is a representation of a path taken by a transaction through a succession of processing modules.
- Think of each transaction as represented by a token-such as a transaction control block that 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

5 Perspective

- Transaction flow graphs are a kind of dataflow graph
- In control flow graphs we defined a link or block as a set of instructions such that if any one of them was executed, all (barring bugs) would be executed.
- For data flow graph in general and transaction flow graphs in particular we change the definition to identify all processes of interest.

2.7.6 Complications

- In simple cases, the transactions have a unique identity from the time they're created to the time they're completed.
- In many systems the transactions can give birth to others, and transactions can also merge.

Births

There are three different possible interpretations of the decision symbol, or nodes with two or more out links. It can be a Decision, Biosis or Mitosis.

1. **Decision:** Here the transaction will take one alternative or the other alternative but not both. (See Figure 3.2 (a))
2. **Biosis:** Here the incoming transaction gives birth to a new transaction, and both transactions continue on their separate paths, and the parent retains its identity. (See Figure 3.2 (b))

COURSE CODE: 17CTU501B

UNIT: III (Transaction-Flow Testing and Data-Flow Testing) BATCH-2017-2020

3. **Mitosis:** Here the parent transaction is destroyed and two new transactions are created.(See Figure 3.2 (c))

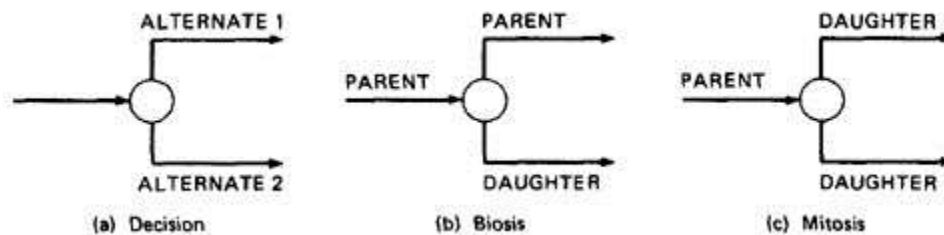


Figure 3.2: Nodes with multiple out links

Mergers

Transaction flow junction points are potentially as troublesome as transaction flow splits. There are three types of junctions: (1) Ordinary Junction (2) Absorption (3) Conjugation

0. **Ordinary Junction:** An ordinary junction which is similar to the junction in a control flow graph. A transaction can arrive either on one link or the other. (See Figure 3.3 (a))
1. **Absorption:** In absorption case, the predator transaction absorbs prey transaction. The prey gone but the predator retains its identity. (See Figure 3.3 (b))
2. **Conjugation:** In conjugation case, the two parent transactions merge to form a new daughter. In keeping with the biological flavor this case is called as conjugation.(See Figure 3.3 (c))

COURSE CODE: 17CTU501B

UNIT: III (Transaction-Flow Testing and Data-Flow Testing) BATCH-2017-2020

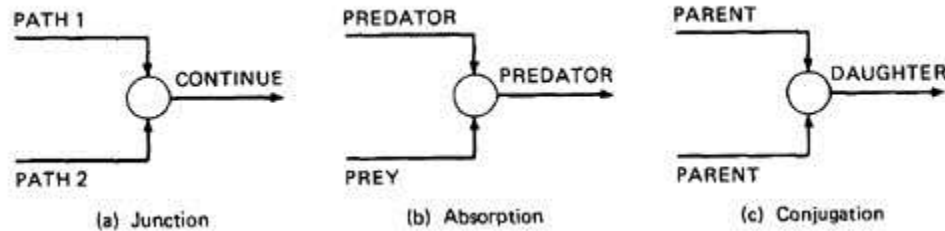


Figure 3.3: Transaction Flow Junctions and Mergers

We have no problem with ordinary decisions and junctions. Births, absorptions, and conjugations are as problematic for the software designer as they are for the software modeler and the test designer; as a consequence, such points have more than their share of bugs. The common problems are: lost daughters, wrongful deaths, and illegitimate births.

3.7.7 Transaction-Flow Structure

- Transaction flows are often ill structured and there is nothing you can do about it.
- Here are some of the reasons
 - It is a model of a process, not just code
 - Parts of the flows may incorporate the behavior of other systems over which we have no control
 - No small part of the totality of transaction flows exists to model error conditions, failures, malfunctions and subsequent recovery actions
 - The number of transactions and the complexity of individual transaction flows grow over time as features are added and enhanced.
 - Systems are build out of modules and the transaction flows result from the interactions of those modules
 - Our models are just that – approximations to reality

COURSE CODE: 17CTU501B

UNIT: III (Transaction-Flow Testing and Data-Flow Testing) BATCH-2017-2020

2.8 Transaction Flow Testing Techniques

2.8.1 Get the Transactions Flows

- Complicated systems that process a lot of different, complicated transactions should have explicit representations of the transactions flows, or the equivalent.
- Transaction flows are like control flow graphs, and consequently we should expect to have them in increasing levels of detail.
- The system's design documentation should contain an overview section that details the main transaction flows.
- Detailed transaction flows are a mandatory pre requisite to the rational design of a system's functional test.

2.8.2 Inspections, Reviews And Walkthroughs

- Transaction flows are natural agenda for system reviews or inspections.
- In conducting the walkthroughs, you should:
 - Discuss enough transaction types to account for 98%-99% of the transaction the system is expected to process.
 - Discuss paths through flows in functional rather than technical terms.
 - Ask the designers to relate every flow to the specification and to show how that transaction, directly or indirectly, follows from the requirements.
- Make transaction flow testing the corner stone of system functional testing just as path testing is the corner stone of unit testing.
- Select additional flow paths for loops, extreme values, and domain boundaries.
- Design more test cases to validate all births and deaths.
- Publish and distribute the selected test paths through the transaction flows as early as possible so that they will exert the maximum beneficial effect on the project.

2.8.3 Path Selection

- Select a set of covering paths (c_1+c_2) using the analogous criteria you used for structural path testing.

COURSE CODE: 17CTU501B

UNIT: III (Transaction-Flow Testing and Data-Flow Testing) BATCH-2017-2020

- Select a covering set of paths based on functionally sensible transactions as you would for control flow graphs.
- Try to find the most tortuous, longest, strangest path from the entry to the exit of the transaction flow.

sensitization

- Most of the normal paths are very easy to sensitize-80% - 95% transaction flow coverage (c1+c2) is usually easy to achieve.
- The remaining small percentage is often very difficult.
- Sensitization is the act of defining the transaction. If there are sensitization problems on the easy paths, then bet on either a bug in transaction flows or a design bug.
- The paths like off-paths, the exception conditions, the path segments, are difficult to sensitize.
- This is because they correspond to error conditions, synchronization problems, overload responses, and other anomalous situations.
- To test these systems perform the following
 - Use Patches – It is a lot easier to fake an error return from another system by a judicious patch than to negotiate a joint test session.
 - Mistune – Test the system sized with grossly inadequate resources
 - Break the Rules – Bypass the database generator and use patches to break any and all rules embodied in the database and system configuration that will help you to go down the desired path.

2.8.4 Path Instrumentation

- Instrumentation plays a bigger role in transaction flow testing than in unit path testing.
- The information of the path taken for a given transaction must be kept with that transaction and can be recorded by a central transaction dispatcher or by the individual processing modules.
- In some systems, such traces are provided by the operating systems or a running log.

2.8.6 Test Databases

COURSE CODE: 17CTU501B

UNIT: III (Transaction-Flow Testing and Data-Flow Testing) BATCH-2017-2020

- About 30% - 40% of the effort of transaction- flow test is the design and maintenance of test database (S).
- Its decided that test databases must be configuration-controlled and centrally administered under a comprehensive design plan.
- In order to avoid the repetition of the previous chaos, it is decided that there will be one comprehensive database that will satisfy all testing needs
- A typical system of a half-million lines of source code will probably need four or five different, incompatible databases to support testing

2.8.7 Execution

- If transaction-flow testing is done for a system of any size, be committed to test execution automation from the start
- If more than a few hundred test cases are required to achieve C1+ C2 transaction flows coverage, do not bother with transaction-flow testing if there is no time and resources to almost completely automate all test execution.

DATA-FLOW TESTING

1. Synopsis

Data-flow testing uses the control flowgraph to explore the unreasonable things that can happen to data (data-flow anomalies).

2. Data-Flow Testing Basics

2.1 Motivation and Assumption

2.1.1 What is it?

Data flow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequence of events related to the status of data objects

2.1.2 Motivation

At least half of contemporary source code consists of data declaration statements –that is statements that define data structures, individual objects, initial or default values and attributes.

2.1.3 New Paradigms-Data Flow Machines

Most computers today are Von Neumann machines. This architecture features interchangeable storage of instruction and data in the same memory units. The Von Neumann Architecture executes one instruction at a time in the following typical micro instruction sequence.

- Fetch instruction from memory.

COURSE CODE: 17CTU501B

UNIT: III (Transaction-Flow Testing and Data-Flow Testing) BATCH-2017-2020

- Interpret instruction.
- Fetch operand(s).
- Process (execute).
- Store Result (perhaps in registers).
- Increment program counter (pointer to next instruction).
- GOTO 1

Massively parallel (multi-instruction, multidata-MIMD) machines, by contrast, have multiple mechanisms for executing steps 1-7 above and can therefore fetch several instruction and/or objects in parallel. They also do arithmetic or logical operations simultaneously on different data objects.

Given L, t and d, solve for Z and H_c .

$\cos C = \cos L \sin t$		
$\tan M = \cot L \cos t$		
$\tan (Z + F) = - \sin L \tan t$		
$\tan F = \cos C \tan (m + d)$		
$\sin H_c = \sin C \sin (m+d)$		
$Z = (Z + F) - F$		
$t1 := \cot L$	$t3 := t3 * t4$	$*/ \cos C /*$
$t2 := \cos t$	$t4 := \tan t1$	$*/ \tan (m+d) /*$
$t3 := t1 * t2$	$t4 := t3 * t4$	$*/ \tan F /*$
$t1 := \tan^{-1} t3$	$t4 := \tan^{-1} t4$	$*/ F /*$
$t1 := t1 + d$	$Z := t2 - t4$	
$t2 := \sin L$	$t3 := \cos^{-1} t3$	$*/ C /*$
$t3 := \tan t$	$t3 := \sin t3$	$*/ \sin C /*$
$t2 := t2 * t3$	$t1 := \sin t1$	$*/ \sin (M + d) /*$
$t2 := \tan^{-1} t2$	$H_c := t1 * t3$	$*/ \sin H_c /*$
$t3 := \cos L$	$H_c := \sin^{-1} H_c$	
$t4 := \sin t$		

Fig: 3.1 von Neumann Navigation Calculation PDL

Fig: 3.1 show the PDL for solving some navigation equations (BOWD77). On a von Neumann machine that has a coprocessor to calculate trigonometric functions the control flowgraph corresponding to fig:3.1 is trivial : it has exactly one link , the implementation shown requires twenty- one steps and four temporary memory locations.

COURSE CODE: 17CTU501B

UNIT: III (Transaction-Flow Testing and Data-Flow Testing) BATCH-2017-2020

2.1.4 The Bug Assumptions

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

2.2 Data Flowgraphs

2.2.1 General

The data flowgraph is a consisting of nodes and directed links (links with arrows on them).

2.2.2 Data Object State and Usage

Data objects can be created, killed and/or used. They can be used in two distinct ways: in a calculation as part of a control flow predicate. The following symbols denote these possibilities:

- d - defined, created, initialized etc,
- k - killed, undefined, released.
- u - used for something.
- c - used in calculation.
- p - used in predicate.

1. Defined- an object is defined explicitly when it appear in data declaration or implicitly (as in FORTRAN) when it appears on the left-hand side of an assignment statement.
2. Killed or undefined - An object is killed or undefined when it is released or otherwise made unavailable, or when its contents or no longer known with certitude.
3. Usage- A variable is used for computation © when it appears on right-hand side of an assignment statement as a pointer as part of a pointer calculation, a file record read or written, and so on.

2.2.3 Data flow anomalies

What is an anomaly may depend on an application for example ,the sequence

A := C + D

IF A > 0 THEN X := 1 ELSE X := -1

A := B + C

Seems reasonable because it corresponds to dpd for variable A, but in the context of some secure systems it might be objectionable because the system doctrine might require several assignments of A to ZERO prior to reuse.

There are nine possible two letter combinations for d, k and u. some are bugs, some or suspicious, and some are okay.

dd – probably harmless but suspicious. Why define the object twice without an intervening

COURSE CODE: 17CTU501B

UNIT: III (Transaction-Flow Testing and Data-Flow Testing) BATCH-2017-2020

usage ?

dk - probably a bug. Why define the object without using it?

du - the normal case the object is defined then used.

kd - normal situation, an object is killed, then redefined.

kk - harmless but probably buggy. Did you want to be sure it as really killed?

ku- a bug the object doesn't exist in the sense that is value is undefined or indeterminate. For example, the loop control value in a FORTAN program after exit from the loop.

ud- usually not a bug because the language permits re-assignments at almost any time.

uk- normal situation

uu- normal situation.

Trailing dash means that nothing after 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

-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.

2.2.4 Data flow Anomaly state graph

Our data flow anomaly model prescribes that an object one of four distinct states:

K – undefined, previously killed, and does not exist.

D- defined but not yet used for anything.

U- has been used for computation or in predicate.

A – anomalous.

COURSE CODE: 17CTU501B

UNIT: III (Transaction-Flow Testing and Data-Flow Testing) BATCH-2017-2020

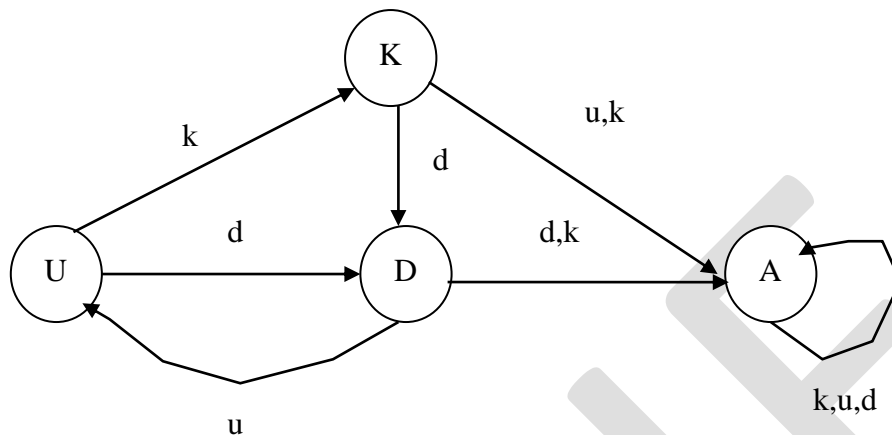


Fig: 3.2 Unforgiving Data Flow Anomaly State graph

FIG – 3.2 shows what is called as the “unforgiving model,” because it holds that once that variable becomes anomalous it can never return to a state of grace.

2.2.5. Static Vs Dynamic Anomaly Detection

Static analysis is an analysis done on source code without actually executing it. Dynamic analysis is done on the fly as the program is being executed and is based on intermediate values that result from the program’s execution.

Barring un-solvability problems, through there are many things for which current notions of static analysis are inadequate.

1. Dead variables – although it is often possible to prove that a variable is dead or alive at a given point in the program. The general problem is unsolvable.
2. Arrays – arrays are problematic in that array is defined or killed as a single object, but reference is to specific location within the array.
3. Records and Pointers – the array problem and the difficulty with pointers is a special case of multipart data structures.
4. Dynamic Subroutine or Function Names in a call – A subroutine or function name is a dynamic variable in a call.
5. False Anomalies – anomalies are specific to paths. Even a “clear bug” such as ku may not be a

COURSE CODE: 17CTU501B

UNIT: III (Transaction-Flow Testing and Data-Flow Testing) BATCH-2017-2020

bug if the path along which the anomaly exists is unachievable.

6. Recoverable Anomalies and Alternate state graphs- the language processor must have a built-in anomaly definition with which you may or may not (with good reason) agree.

7. Concurrency, Interrupts, System issues

2.3 The Data Flow Model

2.3.1 General

Our data-flow model is based on the program's control flow-graph – don't confuse with the program's data flow-graph. we annotate each link with symbols (like Eg: d, k, u, c, p) or sequence of symbols (for example dd, du, ddd) that denote the sequence of the data operation on that link with respect to variable of interest such annotation called link weights.

2.3.2 Components of the Model

Here are the modeling rules,

1. To every statement there is a node, whose name (number) is unique.
2. Exit nodes are dummy nodes placed at the outgoing arrowheads of exit statement s(e.g., END RETURN) to complete the graph.
3. The outlink of simple statements (statements with only one outline) are weighted by the proper sequence of data-flow action for that statement.
4. Predicate nodes (e.g., IF-THEN-ELSE, DO WHILE, CASE) are weighted with the p-use(s) on every outlink, appropriate to that out link.
5. Every sequence of simple statements (e.g., a sequence 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 given variable, then the weight of the link is denoted by the sequence of action 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 most one data- flow action for any variable.

3. Data-Flow Testing Strategies

3.1 General,

Data-flow testing strategies are structural strategies. it is a way of generating a family of test strategies based on a structural characterization of the way test cases are to be defined(i.e., how we pick nodes, links, and/or sequences of nodes or links to be included in a test case) and a functional characterization that test cases must satisfy. In path testing strategies the only structural characteristics used was the raw program-control flow-graph without consideration of what happened on those links. higher-level path testing strategies based, say ,on adjustment link pairs or triplets take more of the control-flow structure into account, but still no other information than is implicit in the control flow-graph, data-flow strategies taken into the account what happen to the data object on the links in addition to the raw connectivity of the graph. data-

COURSE CODE: 17CTU501B

UNIT: III (Transaction-Flow Testing and Data-Flow Testing) BATCH-2017-2020

flow testing strategies are based on selecting test path segments (also called subpath) that satisfy some characteristics of data flows for all data objects.

3.2 Terminology

Some terminology :

- a) A definition clear path segment*(with respect to variable x) is a connected sequence of links such that X is (possibly) defined on the first link and not redefined or killed on any subsequent link of that path segment.
- b) A loop-free path segment is a path segment for which every node is visited at most once.
- c) A simple path segment is a segment in which at most one node is visited twice.
- d) 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.

3.3 The Strategies

3.3.1 Overview

The structural test strategies (FRAN88, RAPP82, and RAPP85) are based on the program's control flow graph. They differ in the extent to which predicate uses and/or computational uses of variables are included in the test set.

3.3.2 All-du paths

The all-du paths (ADUP) strategy is the strongest data-flow testing strategy discussed here. It requires that every du path from every definition of every variable to every use of that definition be exercised under some test. The all-du paths strategy is a criterion, but it does not take as many tests as it might seem at first because any one test simultaneously satisfies the criterion for several definition and uses of several different variables.

3.3.3 All- Uses Strategy

We can reduce the number of test causes by asking that the test set include at least one path segment from every definition to every use that can be reached by that definition – this is called the all-uses (AU) strategy. The strategy is that at least one definition-clear path from every definition of every variable to every use of that definition be exercised under some test.

3.3.4 ALL-P-Uses/Some –c-Uses and All-c-Uses/some-p-uses strategy

The all-p-uses/some-c-uses (APU+C) strategy is defined as follows; for every variable and every definition of that variable, include at least one definition-free path from the definition to every predicate use. The all-c-uses/some-p-uses (ACU-P) strategy reverses the bias; first ensure coverage by computational-use cases and if any definition is not covered by the previously

COURSE CODE: 17CTU501B

UNIT: III (Transaction-Flow Testing and Data-Flow Testing) BATCH-2017-2020

selected paths, add such predicate-use cases as are needed to assure that every definition is included in some test.

3.3.5 All-Definition strategy

The all-definition (AD) strategy asks only that 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.

3.3.6 All-predicate-uses, All-computational uses strategy

The all-predicate-uses (APU) strategy is derived from the APU +C strategy by dropping the requirement that we include a c-use for the variable if there are no p-uses for the variable following each definition. Similarly, the all-computation-uses (ACU) strategy is derived from ACU-P by dropping the requirement that we include a p-use if there are no c-use instances following a definition.

3.3.7 Ordering the strategies

Although ACU+P are stronger than ACU, both are incomparable to the predicate-biased strategies. Note also that “all definitions” is not comparable to ACU or APU.

3.4 Slicing, Dicing, Data flow, and Debugging

3.4.1. General

Testing in a maintenance context is not the same as testing new code-for which most testing theory and testing strategies have been developed. Maintenance testing is in many ways similar to debugging.

3.4.2 Slices and Dices

A (static) slice (WEIS82) is a part of a program(e.g. a selected set of statements) defined with respects to a given variable X (where X is a simple variable or a data vector) and a statement I: it is a set of all statements that could (potentially, under static analysis) affect the value of X at a statement i-where the influence of a faulty statement could result from an improper computational use or predicate use of some other variables at a prior statements. If X is incorrect at a statement-i it follows that bug must be in the program slice for X with respect to i. A program dice (LYLE87)is a part of slice in which all statements which are known to be correct have been removed. In other words, a dice is obtained from a slice by incorporating information obtained through testing or experiment (e.g., debugging). Dynamic slicing (KORE88C) is a refinement of a static slicing in which only statements on achievable paths to the statement in question are included.

COURSE CODE: 17CTU501B

UNIT: III (Transaction-Flow Testing and Data-Flow Testing) BATCH-2017-2020

4. Application, Tools, Effectiveness

Ntafos (NTAF84B) compared random testing, p2 and AU strategies on fourteen of the Kernighan and plauger (KERN76) programs (a set of mathematical programs with known bugs, often used to evaluate test strategies).the experiment had the following outcome:

Strategy	Mean no. test cases	Bugs found (%)
Random testing	35	93.7
Branch testing	3.8	91.6
All uses	11.3	96.3

The second experiment (NTAF84A) on seven similar programs showed the following:

Strategy	Mean no. test cases	Bugs found (%)
Random testing	100	79.5
Branch testing	34	85.5
All uses	84	90.0

Sneed (SNEE86) reports experience with real programs and compares branch coverage effectiveness at catching bugs to data-flow testing criteria. Weyuker (WEYU88A,WEYU90) has published the most through comparison of data flow testing strategies to date. her study is based on

twenty nine programs from the Kernighan and plauger set. Tests were designed using the ASSET testing system (FRAN88).the number of test cases is normalized to the number of binary decision

in the program. just as statement and branch coverage were found to be cost-effective testing strategies, even when unsupported by automation, data-flow testing has been found effective.



KARPAGAM ACADEMY OF HIGHER EDUCATION

DEPARTMENT OF COMPUTER SCIENCE

III B.Sc IT (Batch 2017-2020)

SOFTWARE TESTING (17ITU501B)

PART - A OBJECTIVE TYPE/MULTIPLE CHOICE QUESTIONS

ONLINE EXAMINATIONS

ONE MARKS QUESTIONS

UNIT 3

S.NO	Question	Option 1	Option 2	Option 3	Option 4	Answer
1	_____ Testing is uses the control flow graph to explore the unreasonable things that happen to data	Segment	Node	Data flow	Path	Data flow
2	Data flow testing is the name given to a family of test strategies based on selecting _____ through the program	Node	Path	Link	Error	Path
3	The data flow graph is consisting of _____ and _____	Node and Path	Node and Segment	Link and Segment	Node and DirectedLink	Node and DirectedLink
4	An object is _____ When it appears in data decleration or appears on the left hand side of an assignment statement	Defined implicitly	Defined explicitly	Constant	None of these	Defined explicitly
5	An object is _____ or undefined when it is released or otherwise made unavailable	Not defined	Dead	Killed	Both A and B	killed
6	_____ mens that nothing after point of interest to the exit	Trailing	Timing	Pointing	Exiting	Trailing
7	_____ is an analysis done on source code without actually executing it	Static analysis	Dynamic analysis	valid analysis	Barring analysis	Static analysis
8	_____ is done on the fly as the prrogram is being executing and is based on intermediate values that result from the program's execution	Static analysis	Dynamic analysis	valid analysis	Barring analysis	Dynamic analysis

9	_____ are dummy nodes placed at the outgoing arrowheads of exit statements to compleat the graph	Entry node	Exit node	Constent node	Valid node	Exit node
10	A _____ path segment is a connected sequence of links	Loop free	Simple	Definitio clear	None of these	definition clear
11	A _____ path segment is a path segment for which every node is visited at most once	Loop free	Simple	Definitio clear	None of these	Loop free
12	A _____ path segment is a segment in whicch at most one node is visited twice	Loop free	Simple	Definitio clear	None of these	Simple
13	The _____ strategy is the strongest data-flow testing strategy	all-du paths	non-du paths	Valid-du paths	Strong paths	all-du paths
14	A _____ strategy is reduce the number of test cases by asking that the test set include atleast one path segment from every definition	all-du paths	non-du paths	Valid-du paths	all-uses paths	all-uses paths
15	The _____ strategy asks only that every definition of every variable be covered by atleast one use of that variable	all-du paths	all-Definition	non-du paths	Valid-du paths	all-Definition
16	A _____ is a part of slice in which all statements which are known to be correct have been removed	Slice	Dice	Role	Para	Dice
17	_____ testing is attempt to determine wheather the classification is or is not correct	Statement	path	domain	segment	domain
18	If domain testing is based on _____ then it is a functional test technique	Specification	Qualification	Implementatio n	association	Specification
19	If domain testing is based on _____ then it is a structural test technique	Specification	Qualification	Implementatio n	association	Implementation
20	If domain testing is applied to _____,then predicate intrepretation must be based on actual paths through the routin path	Model	Structure	Link	Path	Structure

21	compound predicate that includes Ors can create_____	Small domains	large domains	concave domains	convex domains	concave domains
22	domains are defined by their_____	nodes	conditions	paths	boundaries	boundaries
23	Every boundary serves atleast _____ different domains	2	3	1	4	2
24	A _____ is a point that does not lie between any two other arbitrary but distinct points of a domain	Boundary point	Extrem point	Boundary point	interior point	Extrem point
25	A _____ point is a point in the domain such that all point within the arbitrary small distance are also in domain	Boundary point	Extrem point	Boundary point	interior point	interior point
26	A _____ - boundary occurs when coefficient in the boundary inequality are wrong	Tilted boundary	Extra boundary	mixed boundary	Shifted boundary	Tilted boundary
27	The bug is a shift up, which converts part of domain B into A processing is _____	Tilted boundary	Extra boundary	mixed boundary	Shifted boundary	Shifted boundary
28	An extra boundary is created by _____	Extra predicate	strain predicate	Shifted predicate	Mixed predicate	Extra predicate
29	An _____ will slice through many different domain and will therefore cause many test failures for the same bug	Tilted boundary	Extra boundary	mixed boundary	Shifted boundary	Extra boundary
30	A _____ boundary will merge different domain and as the extra boundary can, will cause many	Tilted boundary	Extra boundary	mixed boundary	missing boundary	missing boundary
31	A good _____ should have include all interesting domain testing cases	Component test	Path test	Valid test	Invalid test	Component test
32	We can often convert nonlinear boundaries to Equivalent linear boundaries. This is done by applying_____	Debugging the problem	Linearizing transformation	Validation process	None of these	Linearizing transformation
33	_____ boundary set are sets of linear related boundaries	Linear	Non linear	Parallel	None of these	linear

UNIT-IV

SYLLABUS

Domain Testing

Domains and Paths-Domain Testing-Domains and Interface Testing-Domains and Testability

DOMAIN TESTING

1. Synopsis

Programs as input data classifiers: domain testing attempts to determine whether the classification is or is not correct.

2. Domains and Paths

2.1 The model

Domain testing can be based on specifications and/or equivalent implementation information. If domain testing is based on specification, it is a functional test technique; if based on implementations, it is a structural technique. A routine must classify the input and set it moving on the right path. Processing begins with a classifier section that partition the input vector into cases. An invalid input (e.g., value too big) is just a special processing case called “reject” say. The input then passes to a hypothetical subroutine or path that does the processing. In domain testing we focus on the classification aspect of the routine rather than on the calculations.

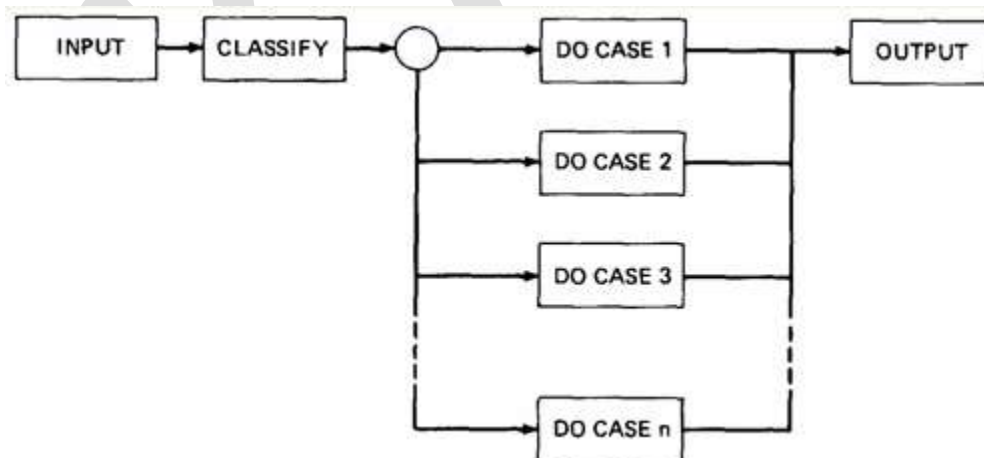


Fig 3.3 Schematic Representation of Domain Testing

2.2 A Domain is a set

An input domain is a set. If the source language supports set definition (e.g. Pascal set types, C enumerated types) less testing is needed because the compiler (compile time and run time) does much of it for us.

The language of set theory is natural for domain testing.

2.3 Domains, paths and predicates

If domain testing is applied to structure, then predicate interpretation must be based on actual paths through the routine that is, based on implementation control flow graph. Conversely, if domain testing is applied to specification, interpretation is based on a specified data flowgraph for the routine.

For every domain there is at least one path through the routine. There may be more than one path if the domain consists of disconnected parts or if the domain is defined by the union of two or more domains. For every boundary there is at least one predicate that specifies what numbers belong to the domain and what numbers don't.

For e.g. in the statement

IF $x > 0$ THEN ALPHA ELSE BETA

We know that numbers greater than zero belong to ALPHA processing domain(s) while zero and smaller numbers belong to BETA domain(s). A domain might be defined by a sequence of predicate-say A,B, and C. first evaluate A and process the A cases, then evaluate B and do B processing, and then evaluate C and finish up. With three binary predicates, there are up to eight (2^3) domains.

To review:

1. A domain for a loop-free program corresponds to a set of number defined over the input vector.
2. For every domain there is at least one path through the routine, along which that domain processing is done.
3. The set of interpreted predicates transverse on that path (i.e. The path's predicate expression) defines the domain's boundary.

2.4 Domain Closure

As in set theory, a domain boundary is closed with respect to a domain if the parts on the boundary belong to the domain. If the boundary points belong to some other domain, the boundary is said to be open.

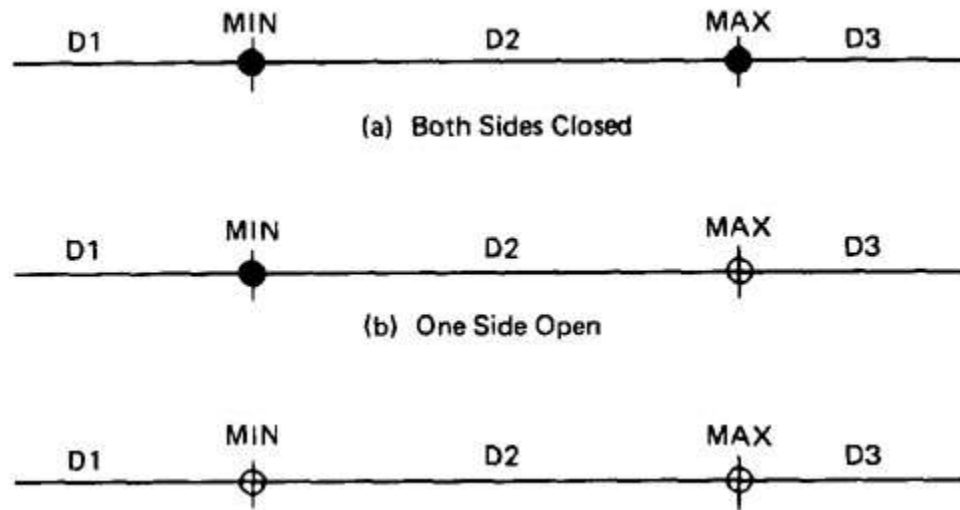


Fig 3.4 Open and Closed Domains

2.5 Domain Dimensionality

Every boundary slices through the input vector space with a dimensionality which is less than the dimensionality of the space. Thus, planes are cut by lines and points, volume by planes, lines and points, and n-spaces by hyperplanes. Spaces of more than three dimensions are called n-spaces. Things that cut through n-spaces are called hyperplanes (not starships). An input array with dimension 100 is not unusual, but it is a 100-dimensional space.

2.6 The Bug Assumptions

An incorrectly implemented domain means that boundary is wrong, which may in turn mean that control-flow predicates are wrong. Many different bugs can result in domain errors. Here is a sample of more common ones:

1. DOUBLE ZERO representation
2. Floating-point zero check

3. Contradictory domains - A contradictory domain specification means that at least two supposedly distinct domain overlap
4. Ambiguous Domain – It means that the union of specified domains is incomplete.
5. Over Specified Domains- The domain can be over loaded with so many conditions that the result is a null domain.
6. Boundary Errors
7. Closure Reversal - The predicate is defined in terms of \geq . The programmer chooses to implement the logical complement and incorrectly uses \leq for the new predicate; i.e. $x \geq 0$ is incorrectly negated as $x \leq 0$.
8. Faulty Logic – Compound predicates (especially) are subject to faulty logic transformation and improper simplification.

2.7 Restriction

2.7.1 General

In testing (other than faulty outcome prediction, improper execution, or other test design and execution bugs) there are no invalid tests – only unproductive tests.

2.7.2 Coincidental Correctness

If the domain is a disconnected mess of small sub domain testing may not be revealing.

2.7.3 Representative Outcome

Domain testing is an example of partition-testing strategies divide the program's input spaces into domain such that all inputs within domain equivalent (not equal, but equivalent) in the sense that any input represents all inputs in that domain. Most test technique, functional or structural, fall under partition testing and therefore make this representative outcome assumption. Another way to say it is that only one functions.

2.7.4 Simple Domain Boundaries and Compound Predicates

Compound predicates that include ORs can create concave domains. Domains that is not simply

connected. These sub domains can be separated, adjacent but not overlapped, partially overlapped, or one can be contained within the other; all of these are problematic.

2.7.5 Functional Homogeneity of Bugs

For linear predicates (i.e. boundary predicates that are linear function) the bug is such that the resulting predicate will still be linear.

2.7.6 Linear vector space

A linear (boundary) predicate is defined by a linear inequality (after interpretation in terms of input variable). * using only the simple relational operators $>$, $>=$, $=$, $<=$, $<$ and $<>$. A more general assumption is that boundaries can be embedded in a linear vector space.

For e.g. the predicate $x^2 + y^2 > a^2$ is not linear in rectangular co-ordinates, but by transforming to polar coordinates we obtain the equivalent linear predicate $r > a$.

2.7.7 Loop-Free Software

If a loop is an overall control loop on transaction, say there's no problem. We 'break' the loop and use domain transaction process. If the loop is definite (that is, if we know on entry exactly how many times it will loop), then domain testing may be useful for the processing within the loop.

Domains are and will be defined by an imperfect iterative process aimed at achieving (user, buyer, voter) satisfaction. The first step in applying the domain testing should be to analyze domain definition in order to get (atleast) consistent and complete domain specifications.

3. Domain Testing

3.1. Overview

The domain-testing strategy is simple, albeit possibly tedious.

1. Domains are defined by their boundaries; therefore, domain testing concentrates test points on or near boundaries.
2. Classify what can go wrong with boundaries, and then define a test strategy for each case. Pick enough points to test for all recognized kinds of boundary errors.
3. Because every boundary serves at least two different domains, test points used to check one domain can also be used to check adjacent domains. Remove redundant test points.

4. Run the tests and by posttest analysis (the tedious part) determine if any boundaries are faulty and if so, how.
5. Run enough tests to verify every boundary of every domain.

3.2. Domain Bugs and How to Test For Them

3.2.1 General

An interior point is a point in the domain such that all points within an arbitrarily small distance (called epsilon neighborhood) are also in domain.

A boundary point is one such that within the epsilon neighborhood there are points both in the domain and not in the domain.

An extreme point is a point that does not lie between any two other arbitrary but distinct points of a (convex) domain.

An on point is a point on the boundary. If the domain boundary is closed, an off point is a point near the boundary but in the adjacent domain.

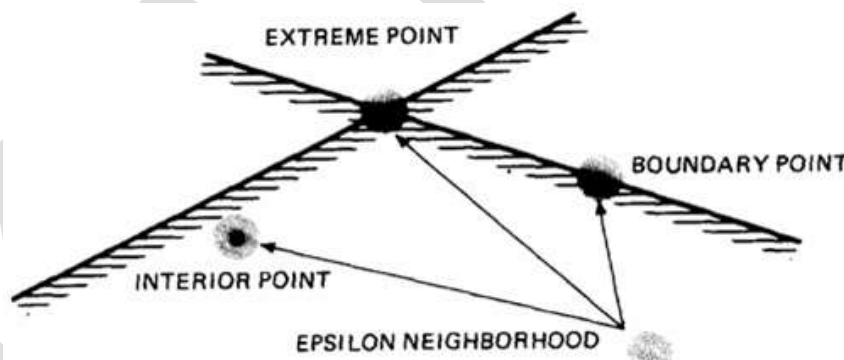


Fig 3.5 Interior, Boundary and Extreme Points

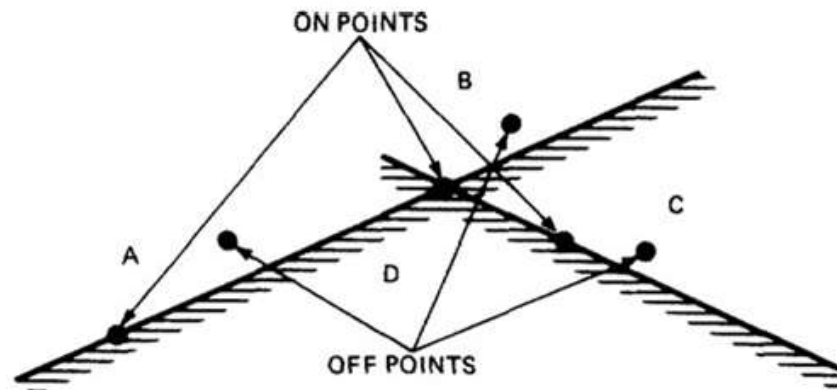
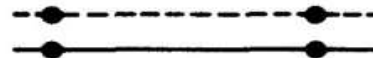


Fig 3.6 On Points and Off Points

SHIFTED BOUNDARIES



TILTED BOUNDARIES



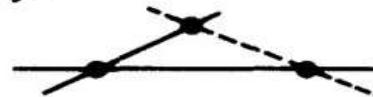
OPEN/CLOSED ERROR



EXTRA BOUNDARY



MISSING BOUNDARY



CORRECT



INCORRECT



Fig 3.7 Generic Domain Bugs

3.2.2 Testing One-Dimensional Domains

The Fig shows possible domain bugs for one dimensional open domain boundary. The closure can be wrong (i.e., assigned to the wrong domain). Or the boundary (a point in the case) can be shifted one way or the other, we can be missing a boundary, or we can have an extra boundary.

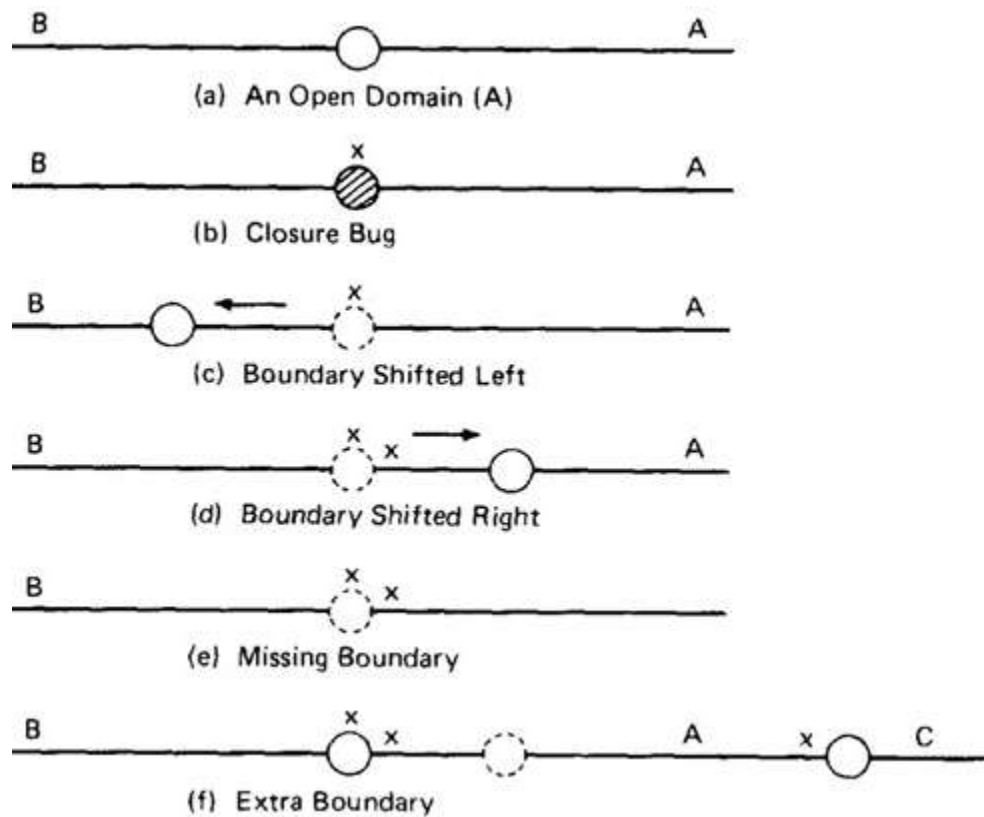


Fig 3.8 One-Dimensional Domain Bugs, Open Boundaries

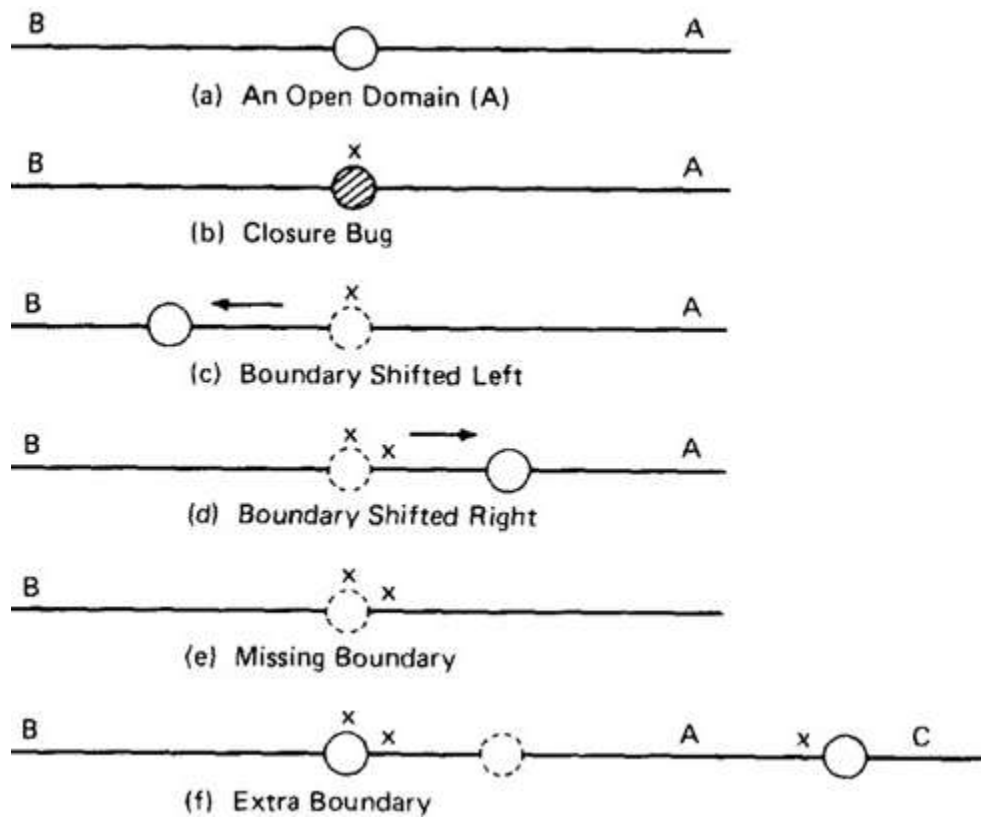


Fig 3.9 One-Dimensional Domain Bugs, Closed Boundaries

The Figure 3.8 shows domain boundary bugs for two dimensional domains A and B are adjacent domains and the boundary is closed with respect to A. which means that it is open with respect to B.

3.2.3 Two Dimensional Domain Bugs

Closure Bug- shows a faulty closure such as might be caused by using a wrong operator.

Shifted Boundary- the bug is a shift up, which converts part of domain B into A processing, denoted by A.

Tilted Boundary- A tilted boundary occurs when co efficient in the boundary inequality are

wrong.

Extra Boundary- an extra boundary is created by an extra predicate. An extra boundary will slice through many different domains and will therefore cause many test failures for the same bug.

Missing Boundary- A missing boundary is created by leaving a boundary predicate out. A missing boundary will merge different domains and as the extra boundary can, will cause many test failure although there is only one bug.

3.2.4 Equality and Inequality Predicates

Equality predicates such as $x + y = 17$ define lower-dimensional domains. For example, if there are two input variables. A two dimensional space, an equality predicate defines a line- a one dimensional domain. Similarly, an equality predicate in three dimensions defines a planner domain.

3.2.5 Random Testing

Domain testing, especially when it incorporates extreme points, has two virtues; it verifies domain boundary efficiently, and many selected test cases (on points an extreme points) corresponds to cases where experience shows programmers have trouble.

3.2.6 Testing n-Dimensional Domains

For domains defined over an n-dimensional input space with p boundary segments, the domain testing strategy generalizes to require at most $(n+1)p$ test points per domain, consisting of n on points and one off point.

3.3. Procedure

The procedure is conceptually straight-forward. It can be done by hand for two domains and few domains.

Identify input variables

Identify variables which appear in domain-defining predicates, such as control-flow predicates.

Interpret all domain predicates in terms of input variables.

For p binary predicates, there are at most $2(P)$ combinations of TRUE –FALSE values and therefore at $2(p)$ domains. Find the set of non-null domains. Each product term (that is, term consisting of predicates joined by ANDs) is a set of linear inequalities that defines a domain or a part of multiply domain.

Solve these inequalities to find all the extreme points of each domain.

Use the extreme points to solve for nearby on points and to locate mid span off points for every domain.

3.4. Variation, Tools, Effectiveness

Variation has been explored that vary the number of on and off points and/or the extreme points. Some specification based tools such as T (PROG88), use heuristic domain-testing. The fact testing is tool –intensive should not be barrier to its effective exploitations.

4. Domains and Interface Testing

4.1 General

Integration testing is testing the correctness of the interface between two otherwise correct components. For a single variable, the domain span is the set of numbers between (including) the smallest value and largest value. For every input variable we need (atleast); compatible domain span and incompatible closures.

4.2 Domains and range

The set of output values produced by a function, in correct with the domain, which is the set of input values over which the function is defined.

An interface test consists of exploring the correctness of the following mappings

caller domain \rightarrow caller range (caller unit test)

caller range \rightarrow called domain (integration test)

called domain → called range (called unit test)

4.3 Closure Compatibility

Fig 3.10 shows the four ways in which the caller's range closure and called's domain closure can agree. The thick line means closed and the thin line means open.

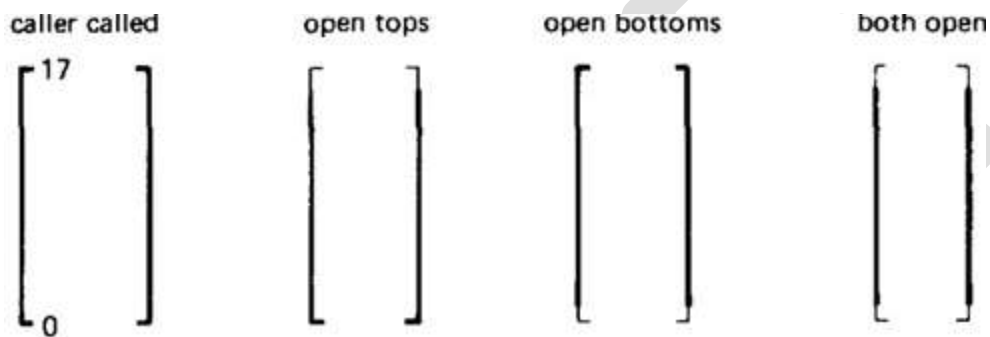


Fig 3.10 Range/Domain Closure Compatibility

Fig 3.11 shows the twelve different ways the caller and the called can disagree about closure. The four cases in which a caller boundary is open and the called is closed (marked with a '?') are probably not buggy.

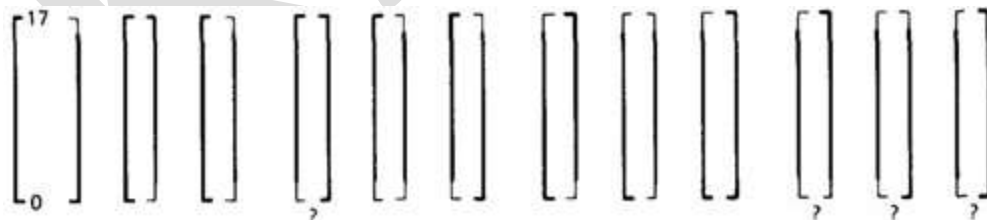


Fig 3.11 Equal-Span Range/Domain Compatibility Bugs

4.4. Span Compatibility

Fig 3.12 shows three possible harmless span incompatibilities. consider, for example, a square-root routine that accepts negative numbers and provides an imaginary square root for them. The

routine is used by many callers; some require complex number answer and some don't. This kind of span incompatibility is a bug only if the caller expects the called routine to validate the called numbers for the callers. If that's so, there are values that can be included the call, which from the caller's point of view are invalid, but for which the called routine will not provide an error response.

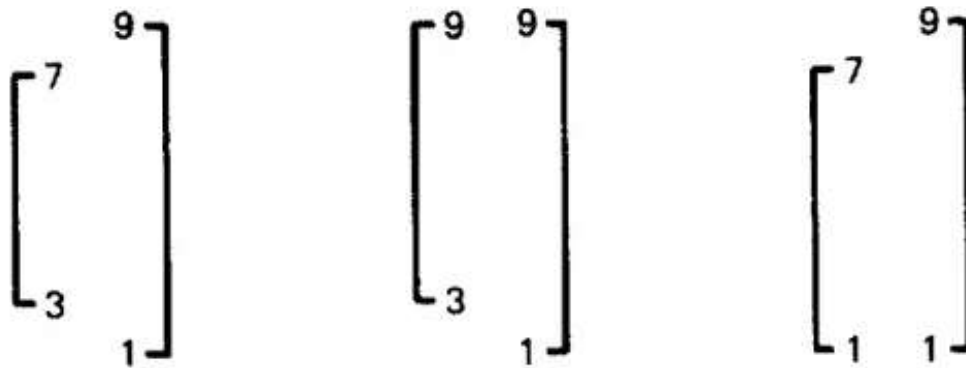


Fig 3.12 Harmless Range/Domain Span Incompatibility Bug. (Caller Span Is Smaller Than Called.)

[Fig 3.13a](#) shows the opposite situation, in which the called routine's domain has a smaller span than the caller expects. All of these examples are buggy. In [Fig 3.12 b](#) the ranges and domains don't line up; hence good values are rejected, bad values are accepted, and if the called routine isn't robust enough, we have crashes. [Fig 3.13 c](#) combines these notions to show various ways we can have holes in the domain: these are all probably buggy.

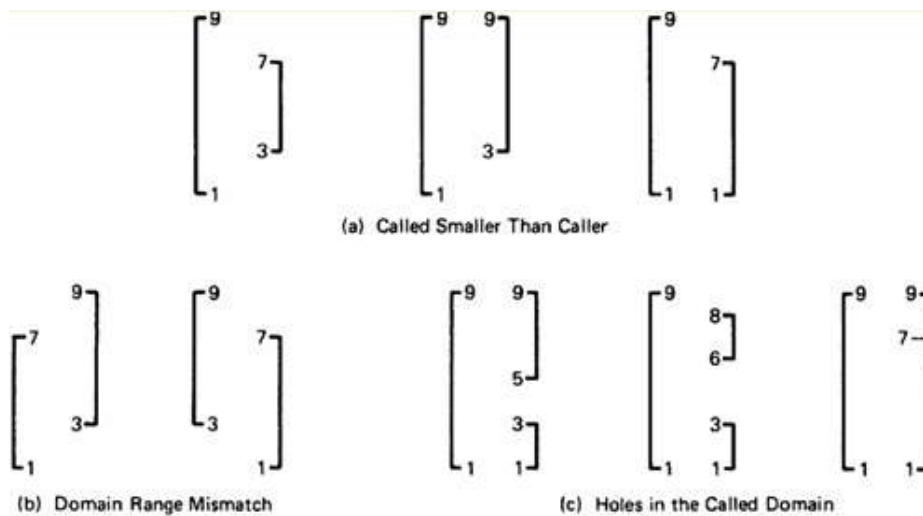


Fig 3.13 Buggy Range/Domain Mismatches

4.5 Interface Range/Domain compatibility Testing

Test every input variable independently of other input variables to conform compatibility of the caller's range and the called routine's domain span and closure of every domain defined for that variable for sub-routine's that classified input as "valid/invalid" the called routine's domain span and closure (for valid cases) is usually broader than the caller's span and closure. if domains divide processing into several cases, you must use normal domain –testing ideas and look for exact range/domain boundary matches.

4.6 Finding the Values

A good component test should have included all interesting domain-testing cases, and as a consequence there's no need to repeat the work. Those test cases are the values for which you must find the input values of the caller. If the caller's domains are linear, then this is another exercise in solving inequalities. Some things to consider:

1. The solution may not be unique
2. There may be no for the specific points you need
3. Find and evaluate an inverse function for the caller. Note "an inverse" rather than "the inverse".

4. Unless we are a mathematical whiz we won't be able to do this without tools for more than one variable at a time.

5. Domains and Testability

5.1 General

The best way to domain testing is to avoid it by making things so simple that it isn't needed.

5.2 Linearizing transformation

We can often convert nonlinear boundaries to equivalent linear boundaries. This is done by applying linearizing transformation.

1. Polynomials

A boundary is specified by a polynomial or multinomial in several variables.

2. Logarithmic transforms

Products such as XYZ can be linearized by substitute $u=\log(x)$, $v=\log(y)$, $w=\log(z)$. The original predicate ($xyz > 17$, say) now becomes $u+v+w > 2.83$.

3. More general transforms

Other linearizable forms include $x/(ax+b)$ and $ax/(b)$. We can also linearize (approximately) by using the Taylor series expansion of nonlinear functions which yields an infinite polynomial.

5.3 Coordinate Transformation

Nice boundaries come in parallel sets. Parallel boundary set are sets of linear related boundaries: that is, they differ only by the value of constant. Finding such sets is straightforward. pick a variable, say x . it has a co-efficient; in inequality i . divided each inequality by it's x coefficient so that the coefficient of the transformed set of inequalities is unity for variable x . if to inequality are parallel, then all the coefficients will be the same and they will differ only by a constant. from the set of nonparallel inequality. a subset that can, by suitable coordinate transformation, be converted into a set of orthogonal inequality. the reason for doing this to

obtain a new set of variables in-terms of which the inequalities can be tested one at a time, independently of the other inequalities.

5.4 A Canonical Program Form

1. Input the data
2. Applying linearizing transforms to as many predicates as possible.
3. Transform to an orthogonal coordinate system.
4. For each set of hyperplanes in the orthogonal space, determine case by table lookup by an efficient search procedure (e.g., binary halving) to put the value of that variable into the right bin.
5. Test the remaining inequalities.
6. Direct the program to correct case processing routine by a table lookup or by a tree control-flow predicates based on the case number for each dimension.

Testing is clearly divided into the following: testing the predicates and coordinate transformations, testing the individual case selection, testing the control flow and then testing the case processing.

5.5 Great insights

Sometimes programmer's have great insights into programming problems that result in much simpler programs than one might have expected. Insights can make a tough programming problem easy; many of them can be explained in terms of a judicious change to a new coordinate system in which the problem becomes easy. There's ample precedence for such things in other engineering disciplines—a good coordinate systems can break the back of many tough problems.



KARPAGAM ACADEMY OF HIGHER EDUCATION

DEPARTMENT OF COMPUTER SCIENCE

III B.Sc IT (Batch 2017-2020)

SOFTWARE TESTING (17ITU501B)

PART - A OBJECTIVE TYPE/MULTIPLE CHOICE QUESTIONS

ONLINE EXAMINATIONS

ONE MARKS QUESTIONS

UNIT 4

S.NO	Question	Option 1	Option 2	Option 3	Option 4	Answer
1	Fundamental problems in our most popular metric are seemingly obvious _____	process of code	functions of code	lines of code	Parameters of code	Lines of code
2	_____ must be objective in the same that the measurement process is algorithmic and will yields the same result no matter who applies it	Ruler	scale	non-metrics	metrics	metrics
3	A useful metric can be calculated _____ for all programs to which we apply it.	randomly	uniquely	dynamically	none of these	uniquely
4	The metrics are classified into _____ types	2	5	3	4	3
5	_____ is based on measuring properties of program or specification text without interpreting what the text means	structural metrics	hybrid metrics	linguistic metrics	parallel metrics	linguistic metrics
6	_____ is based on structural relations between objects in the program	structural metrics	hybrid metrics	linguistic metrics	parallel metrics	structural metrics
7	_____ is bassed on some combination of structural and linguistic properties of program	structural metrics	hybrid metrics	linguistic metrics	parallel metrics	hybrid metrics
8	_____ is to count the coding of a program and use that number as a measure of complexity	number of errors	number of lines	both a and b	none of these	number of lines

9	The most solid confirmation of the bug predication equation is by _____	lipow	lipwo	lipo	none of these	lipow
10	_____ is strongly typed languages that force the explicit declarartion of all types and prohibit mixed typed operation	operator types	data type distinction	data type declaration	both B and C	data type distinction
11	_____ is a structural prototype and therefore the criticism applies to all linguistic metric	operator types	call depth	depth call	structure call	call depth
12	_____ in a programming language is the basic syntactic unit from which programs are constructed	token	keyword	identifier	none of these	token
13	structural metric take the opposite view point of _____	structural metrics	hybrid metrics	parallel metrics	linguistic metrics	linguistic metrics
14	McCabe's cyclomatic complexity metric is defined as _____	$M=L-M+2P$	$M=N-L+2P$	$M=L-N+2P$	none of these	$M=L-N+2P$
15	A _____ statement in a language such as FORTRAN can contain the compound predicate of the form	looping stateement	jump statement	decision statement	case statement	decision statement
16	_____ are introduced as an algebric representation of sets of paths in a graph	path expression	segment expression	flow expression	none of these	path expression
17	Path name is also called as _____	base product	path product	identification of product	none of these	path product
18	The _____ denotes the path in parallel between two nodes	base product	path product	identification of product	path sum	path sum
19	_____ can be understood as an infinite set of parallel paths	loops	segments	transaction	none of these	loops
20	The _____ is the fundamental step of the reduction algorithm	parallel term	cross term step	direct term	indirect term	cross term step

21	A _____ is one that can be reduced to a single link by successive application of the transformation	conditional flow graph	control flow graph	structured flow graph	none of these	structured flow graph
22	The processing time for the link is denoted by _____	S	M	H	T	T
23	The processing time component consists of _____ parts	2	3	4	5	2
24	push and pop are the _____ operations	arithmetic operation	Complementary operations	logical operation	none of these	Complementary operations
25	A _____ table is needed to interpret the weight addition and multiplication operation	arithmetic table	complementary table	logical table	none of these	arithmetic table
26	_____ theorem can be easily generalized to cover sequence of greater length than two character	Huang's theorem	dennis theorem	walkman theorem	none of these	Haung's thorem
27	_____ that denotes all the possible sequence of operators in the graphs	irregular expression	segment expression	transaction expression	regular expression	regular expression
28	Combine all _____ by multiplying their path expression	segment links	transaction links	serial links	process links	serial links

UNIT-V SYLLABUS

Logic-Based Testing and State Graphs

Motivational Overview-Decision Tables-Path Expressions Again-KV Charts-Specifications
State Graphs-Good State Graphs and Bad-State Testing

2. Motivational Overview

2.1. Programmers and Logic

Boolean algebra is to logic as arithmetic is to mathematics. Without it, the tester or programmer is cut off from many test and design techniques and tools that incorporate those techniques.

2.2. Hardware Logic Testing

Hardware logic test design, are intensely automated. Many test methods developed for hardware logic can be adapted to software logic testing

2.3. Specification Systems and Languages

Boolean algebra (also known as the **sentential calculus**) is the most basic of all logic systems. Higher-order logic systems are needed and used for formal specifications

2.4. Knowledge-Based Systems

The **knowledge-based system** (also **expert system** or “artificial intelligence” system) has become the programming construct of choice for many applications that were once considered very difficult. Knowledge-based systems incorporate knowledge from a knowledge domain such as medicine, law, or civil engineering into a database. The data can then be queried and interacted with to provide solutions to problems in that domain.

2.5. Overview

- Start with **decision tables** they are extensively used in business data processing.
- Review of Boolean algebra

- The **Karnaugh-Veitch** diagram - Without it, Boolean algebra is tedious and error-prone

3. Decision Tables

3.1. Definitions and Notation

It consists of four areas called the **condition stub**, the **condition entry**, the **action stub**, and the **action entry**. 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 "I" means that the condition plays no part in the rule, or it is **immaterial** to that rule. The **action stub** names the actions the routine will take or initiate if the rule is satisfied. If the action entry is "YES," the action will take place; if "NO," the action will not take place

		CONDITION ENTRY			
		RULE 1	RULE 2	RULE 3	RULE 4
CONDITION STUB	CONDITION 1	YES	YES	NO	NO
	CONDITION 2	YES	I	NO	I
	CONDITION 3	NO	YES	NO	I
	CONDITION 4	NO	YES	NO	YES
ACTION STUB	ACTION 1	YES	YES	NO	NO
	ACTION 2	NO	NO	YES	NO
	ACTION 3	NO	NO	NO	YES
		ACTION ENTRY			

Fig 5.2 An Example of a Decision Table

- Action 1 will be taken if predicates 1 and 2 are true and if predicates 3 and 4 are false (rule 1), or if predicates 1, 3, and 4 are true (rule 2).
- Action 2 will be taken if the predicates are all false, (rule 3).
- Action 3 will take place if predicate 1 is false and predicate 4 is true (rule 4).

	RULE 5	RULE 6	RULE 7	RULE 8
CONDITION 1	1	NO	YES	YES
CONDITION 2	1	YES	1	NO

CONDITION 3	YES	1	NO	NO
CONDITION 4	NO	NO	YES	1
DEFAULT ACTION	YES	YES	YES	YES

Table 5.1 The Default Rules

In addition to the stated rules, therefore, we also need a **default rule** that specifies the default action to be taken when all other rules fail.

3.2. Decision-Table Processors

The decision table's translator checks the source decision table for consistency and completeness and fills in any required default rules. The usual processing order in the resulting object code is, first, to examine rule 1. If the rule is satisfied, the corresponding action takes place. Otherwise, rule 2 is tried. This process continues until either a satisfied rule results in an action or no rule is satisfied and the default action is taken

3.3. Decision Tables as a Basis for Test Case Design

if a program's logic is to be implemented as decision tables, decision tables should also be used as a basis for test design

The use of a decision-table model to design tests is warranted when:

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

3.4. Expansion of Immaterial Cases

Improperly specified **immaterial entries (I)** cause most decision-table contradictions. If a

condition's truth value is immaterial in a rule, satisfying the rule does not depend on the condition. It doesn't mean that the case is impossible. For example,

Rule 1: "If the persons are male and over 30, then they shall receive a 15% raise."

Rule 2: "But if the persons are female, then they shall receive a 10% raise."

The above rules state that age is material for a male's raise, but immaterial for determining a female's raise. No one would suggest that females either under or over 30 are impossible

	RULE 2		RULE 4			
	RULE 2.1	RULE 2.2	RULE 4.1	RULE 4.2	RULE 4.3	RULE 4.4
CONDITION 1	YES	YES	NO	NO	NO	NO
CONDITION 2	YES	NO	YES	YES	NO	NO
CONDITION 3	YES	YES	YES	NO	NO	YES
CONDITION 4	YES	YES	YES	YES	YES	YES

Table 5.2. The Expansion of an Inconsistent Specification.

Rules 1 and 2 are contradictory because the expansion of rule 1 via condition 2 leads to the same set of predicate truth values as the expansion of rule 2 via condition 3. Therefore action 1 or action 2 is taken depending on which rule is evaluated first.

3.5. Test Case Design

Test case design by decision tables begins with examining the specification's consistency and completeness. Once the specifications have been verified, 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.
2. It is not usually possible to change the order in which the predicates are evaluated because that order is built into the program
3. It is not usually possible to change the order in which the rules are evaluated because that order is built into the program, but if the implementation allows the rule evaluation order to be modified, test different orders for the rules by pairwise interchanges.

4. Identify the places in the routine where rules are invoked. Identify the places where actions are initiated.

3.6. Decision Tables and Structure

Decision tables can also be used to examine a program's structure

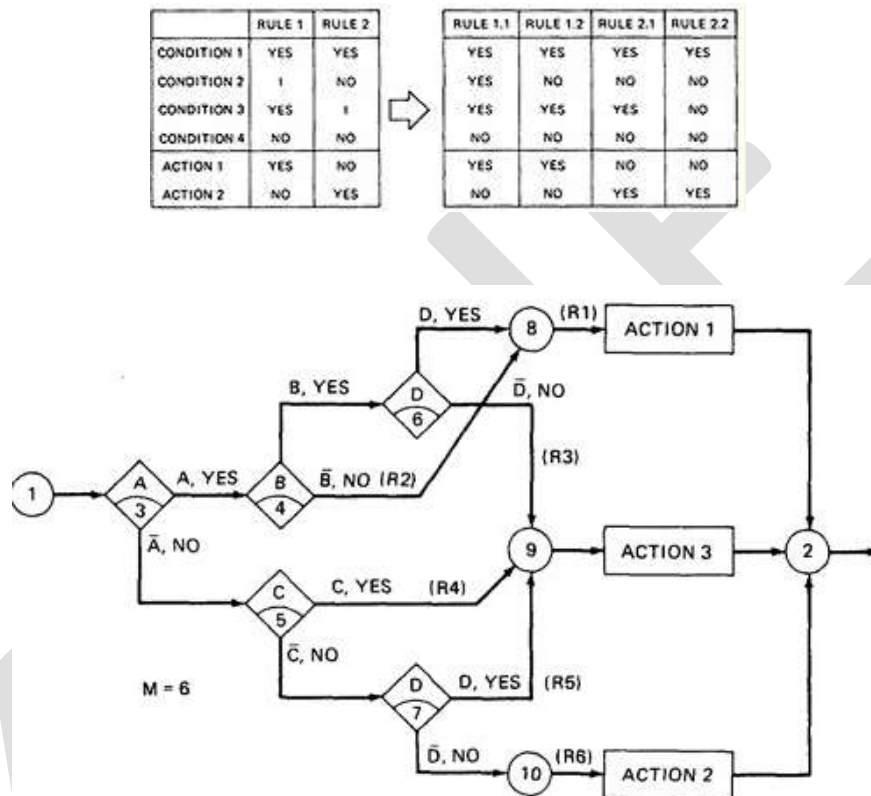


Fig 5.3. A Sample Program

	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	1	1	1
CONDITION C	1	1	1	YES	NO	NO
CONDITION D	YES	1	NO	1	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	

[Table 5.3. The Decision Table](#)

Sixteen cases are represented in [Table 10.5](#), and no case appears twice. Consequently, the flowgraph appears to be complete and consistent. As a first check, before you look for all sixteen combinations, count the number of Y's and N's in each row. They should be equal.

4. Path Expressions Again

4.1. General

4.1.1. The Model

In logic-based testing we focus on the truth values of control flow predicates.

4.1.2. Predicates and Relational Operators

A **predicate** is implemented as a process whose outcome is a truth-functional value. Predicates are based on **relational operators**, of which the arithmetic relational operators are merely the most common.

4.1.3. Case Statements and Multivalued Logics

Logic-based testing is restricted to binary predicates. If you have case statements, you have to analyze things one case at a time if you're to use these methods

4.1.4. What Goes Wrong with Predicates

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

1. The wrong relational operator is used: e.g., > instead of <=.
2. The predicate expression of a compound predicate is incorrect: e.g., A + B instead of AB.

3. The wrong operands are used: e.g., $A > X$ instead of $A > Z$.

4. The processing leading to the predicate (along the predicate's interpretation path) is faulty.

4.1.5. Overview

- convert the path expressions into Boolean algebra, using the predicates' truth values as weights
- examine the logical sum of those expressions for consistency and ambiguity
- consider a hierarchy of logic-based testing strategies and associated coverage concepts

4.2. Boolean Algebra

4.2.1. Notation

Let's take a structural viewpoint for the moment and review the steps taken to get the predicate expression of a path.

1. Label each decision with an uppercase letter that represents the truth value of the predicate. The YES or TRUE branch is labeled with a letter and the NO or FALSE branch with the same letter overscored.
2. The truth value of a path is the product of the individual labels. Concatenation or products mean "AND."
3. If two or more paths merge at a node, the fact is expressed by use of a plus sign (+) which means "OR."

4.2.2. The Rules of Boolean Algebra

Boolean algebra has three operators:

RULES								
	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}C$	$\bar{A}BC$	$A\bar{B}\bar{C}$	$A\bar{B}C$	ABC	$A\bar{B}C$	ABC
CONDITION A	NO	NO	NO	NO	YES	YES	YES	YES
CONDITION B	NO	NO	YES	YES	YES	YES	NO	NO
CONDITION C	NO	YES	YES	NO	NO	YES	YES	NO
ACTION 1	YES	NO	NO	NO	YES	YES	YES	YES
ACTION 2	YES	YES	YES	YES	YES	YES	NO	NO
ACTION 3	YES	YES	YES	NO	NO	YES	YES	NO

- × meaning AND. Also called multiplication. A statement such as AB means “A and B are both true.” This symbol is usually left out as in ordinary algebra.
- + meaning OR. “A + B” means “either A is true or B is true or both.”
- \bar{A} meaning NOT. Also negation or complementation.

Laws of Boolean algebra

$$1. \frac{A + A}{\bar{A} + \bar{A}} = \frac{A}{\bar{A}}$$

$$2. A + 1 = 1$$

$$3. A + 0 = A$$

$$4. A + B = B + A$$

$$5. A + \bar{A} = 1$$

$$6. \frac{A\bar{A}}{\bar{A}\bar{A}} = \frac{A}{\bar{A}}$$

$$7. A \times 1 = A$$

$$8. A \times 0 = 0$$

$$9. AB = BA$$

$$10. A\bar{A} = 0$$

$$1. \bar{\bar{A}} = A$$

$$2. \bar{0} = 1$$

$$3. \bar{1} = 0$$

$$4. \overline{A + B} = \bar{A}\bar{B}$$

$$5. \overline{AB} = \bar{A} + \bar{B}$$

$$6. A(B + C) = AB + AC$$

$$7. (AB)C = A(BC)$$

$$8. (A + B) + C = A + (B + C)$$

$$9. A + \bar{A}B = A + \bar{B}$$

$$10. A + AB = A$$

If something is true, saying it twice doesn't make it truer, ditto for falsehoods.

If something is always true, then "either A or true or both" must also be universally true.

Commutative law.

If either A is true or not-A is true, then the statement is always true.

A statement can't be simultaneously true and false.

"You ain't not going" means you are. How about, "I ain't not never going to get this nohow,"?

Called "De Morgan's theorem or law."

Distributive law.

Multiplication is associative.

So is addition.

Absorptive law.

4.2.3. Examples

The path expressions of Section 4.2 can now be simplified by applying the rules

$N6 = A + \overline{A}\overline{B}\overline{C}$	
$= A + \overline{B}\overline{C}$: Use rule 19, with "B" = $\overline{B}\overline{C}$.
$N8 = (N6)B + \overline{A}B$	
$= (A + \overline{B}\overline{C})B + \overline{A}B$: Substitution.
$= AB + \overline{B}\overline{C}B + \overline{A}B$: Rule 16 (distributive law).
$= AB + \overline{B}\overline{C}B + \overline{A}B$: Rule 9 (commutative multiplication).
$= AB + 0C + \overline{A}B$: Rule 10.
$= AB + 0 + \overline{A}B$: Rule 8.
$= AB + \overline{A}B$: Rule 3.
$= (A + \overline{A})B$: Rule 16 (distributive law).
$= 1 \times B$: Rule 5.
$= B$: Rules 7, 9.

4.2.4. Paths and Domains

Consider a loop-free entry/exit path and assume for the moment that all predicates are simple. If a literal appears twice in a product term then not only can one appearance be removed but it also means that the decision is redundant. If a literal appears both barred and unbarred in a product term, then by rule 10 the term is equal to zero, which is to say that the path is unachievable.

A product term on an entry/exit path specifies a domain because each of the underlying predicate expressions specifies a domain boundary over the input space. Because the predicates are compound, the Boolean expression corresponding to the path will be (after simplification) a sum of product terms. Because this expression was derived from one path, the expression also specifies a domain. However, the domain now need not be simply connected

4.2.5. Test Case Design

Although it is possible to have ambiguities and contradictions in a specification (given, say, as a list of conditions), it is not possible for a program to have contradictions or ambiguities if:

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

The set of paths used to reach any point of interest in the flowgraph (such as the exit) can be characterized by an increasingly more thorough set of test cases:

1. Simplest—Use any prime implicant in the expression to the point of interest as a basis for a path.

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—Because in general, the truth value of a predicate is obtained by interpreting the predicate, its value may depend on the path taken to get there. Do every term by every path to that term.

4.3. Boolean Equations

Loops complicate things because we may have to solve a Boolean equation to determine what predicate-value combinations lead to where. Furthermore, the Boolean expression for the end point does not necessarily equal 1

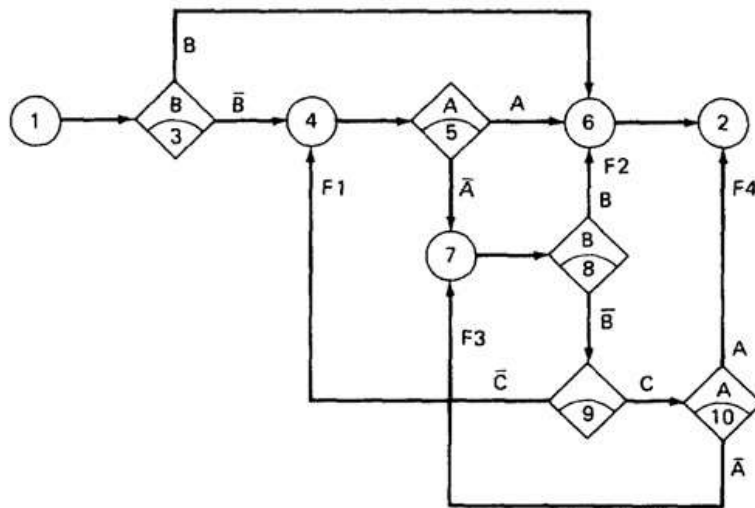


Fig 5.4. A Flowgraph with Loops.

Assign a name to any node. It's usually convenient to give names to links. The names represent the Boolean expression corresponding to that link

$$\begin{aligned}
 N4 &= \overline{B} + F1 \\
 F1 &= \overline{B} \overline{C} N7 \\
 N4 &= \overline{B} + \overline{B} \overline{C} N7 \\
 &= \overline{B} \\
 N6 &= B + AN4 \\
 &= B + \overline{A} B \\
 &= A + B \\
 N7 &= \overline{A} N4 + F3 \\
 &= \overline{A} \overline{B} + F3 \\
 F3 &= N7 \overline{B} \overline{C} \overline{A} \\
 N7 &= \overline{A} \overline{B} + \overline{A} \overline{B} \overline{C} N7 \\
 &= \overline{A} \overline{B} \\
 N2 &= N6 + F4 \\
 &= A + B + F4 \\
 F4 &= \overline{A} \overline{B} \overline{C} N7 \\
 N2 &= A + B + \overline{A} \overline{B} \overline{C} N7 \\
 &= A + B
 \end{aligned}$$

The fact that the expression for the end point does not reduce to 1 means that there are predicate-value combinations for which the routine will loop indefinitely. If the predicate values are independent of the processing, this routine must loop indefinitely.

5. KV CHARTS

5.1. The Problem

The Karnaugh-Veitch chart (this is known by every combination of “Karnaugh” and/or “Veitch” with any one of “map,” “chart,” or “diagram”) reduces Boolean algebraic manipulations to graphical trivia

5.2. Simple Forms

[Fig 5.5](#) shows all the Boolean functions of a single variable and their equivalent representation as a KV chart. The charts show all possible truth values that the variable A can have. The heading

above each box in the chart denotes this fact. A “1” means the variable’s value is “1” or TRUE. A “0” means that the variable’s value is 0 or FALSE.

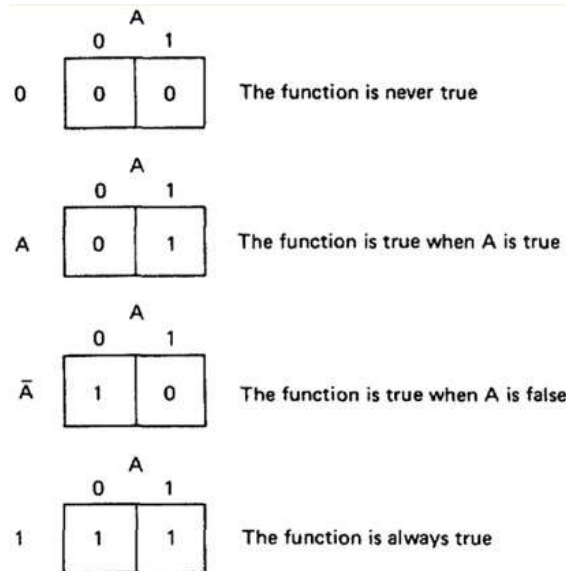


Fig 5.5. KV Charts for Functions of a Single Variable.

[Fig 5.6](#) 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

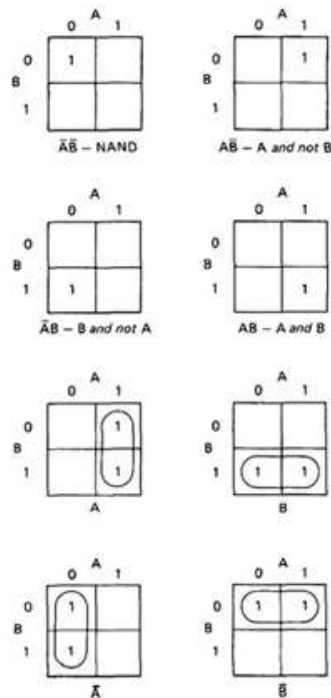


Fig 5.6. Functions of Two Variables.

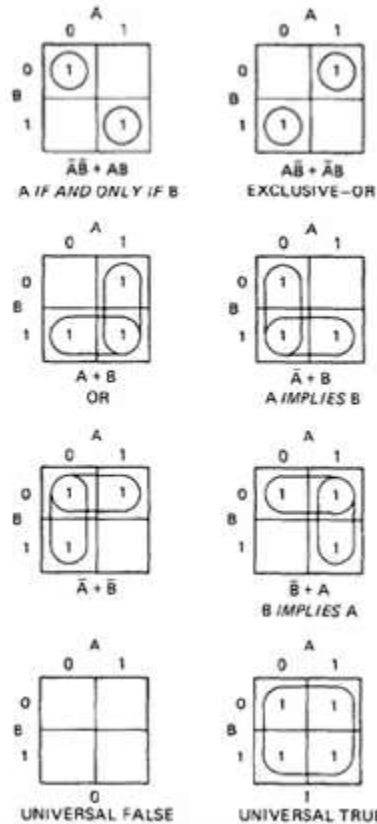


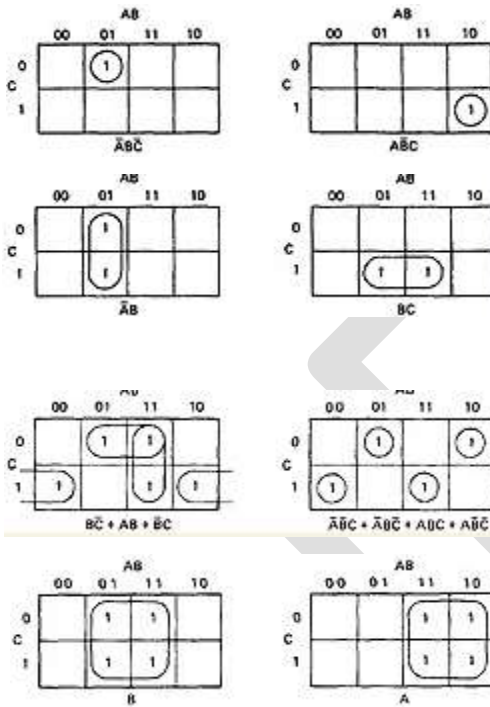
Fig 5.7. More Functions of Two Variables.

Since n variables lead to 2^n combinations of 0 and 1 for the variables, and each such combination (box) can be filled or not filled, leading to 2^{2^n} ways of doing this

5.3. Three Variables

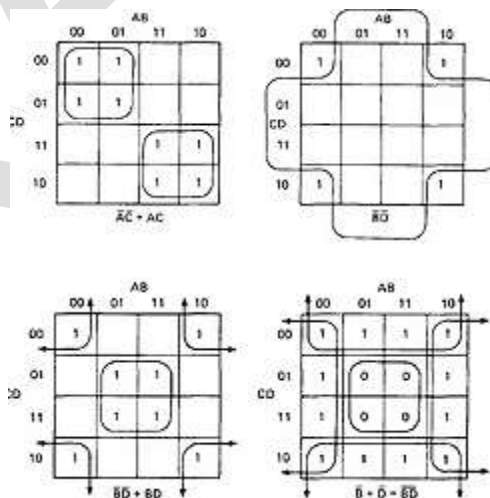
KV charts for three variables are shown below. As before, each box represents an elementary term of three variables with a bar appearing or not appearing according to whether the row-column heading for that box is 0 or 1

A three-variable chart can have groupings of 1, 2, 4, and 8 boxes. A few examples shown below



5.4. Four Variables and More

The same principles hold for four and more variables. A four-variable chart and several possible adjacencies are shown below. Adjacencies can now consist of 1, 2, 4, 8, and 16 boxes, and the terms resulting will have 4, 3, 2, 1, and 0 literals in them respectively:



6. SPECIFICATIONS

6.1. General

The procedure for specification validation is straightforward:

1. Rewrite the specification using consistent terminology.
2. Identify the predicates on which the cases are based. Name them with suitable letters, such as A, B, C.
3. Rewrite the specification in English that uses only the logical connectives AND, OR, and NOT, however stilted it might seem.
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 expressions in a KV chart and check for consistency. If the specifications are consistent, there will be no overlaps, except for cases that result in multiple actions.
7. Enter the default cases and check for consistency.
8. If all boxes are covered, the specification is complete.
9. If the specification is incomplete or inconsistent, translate the corresponding boxes of the KV chart back into English and get a clarification, explanation, or revision.
10. If the default cases were not specified explicitly, translate the default cases back into English and get a confirmation.

6.2. Finding and Translating the Logic

We cast the specifications into sentences of the following form:

“IF predicate THEN action.”

The predicates are written using the AND, OR, and NOT Boolean connectives.

IF A AND B AND C, THEN A1,

IF C AND D AND F, THEN A1,

IF A AND B AND D, THEN A2,

...

Now identify all the NOTs, which can be knotty because some sentences may have the form

$A + B + C$ or \overline{ABC} . Put phrases in parentheses if that helps to clarify things.

6.3. Ambiguities and Contradictions

There is an ambiguity, probably related to the default case. You might get contradictory answers, in which case, you may have to rephrase your question or, better yet, lay out all the combinations in a table and ask for a resolution of the ambiguities. There are several boxes that call for more than one action

6.4. Don't-Care and Impossible Terms

1. Identify all "impossible" and "illogical" cases and confirm them.
2. Document the fact that you intend to take advantage of them.
3. Fill out the KV chart with the possible cases and then fill in the impossible cases. Use the combined symbol ϕ , which is to be interpreted as a 0 or 1, depending on which value provides the greatest simplification of the logic. These terms are called **don't-care terms**, because the case is presumed impossible, and we don't care which value (0 or 1) is used.

STATES, STATE GRAPHS

1. State Graphs

1.1. States

We define "state" as: "A combination of circumstances or attributes belonging for the time being to a person or thing."

A person's checkbook can have the following states with respect to the bank balance:

1. Equal
2. Less than
3. Greater than

A word processing program menu can be in the following states with respect to file

manipulation:

1. Copy document
2. Delete document
3. Rename document
4. Create document
5. Compress document
6. Copy disc
7. Format disc
8. Backup disc
9. Recover from backup

States are represented by nodes. States are numbered or may be identified by words

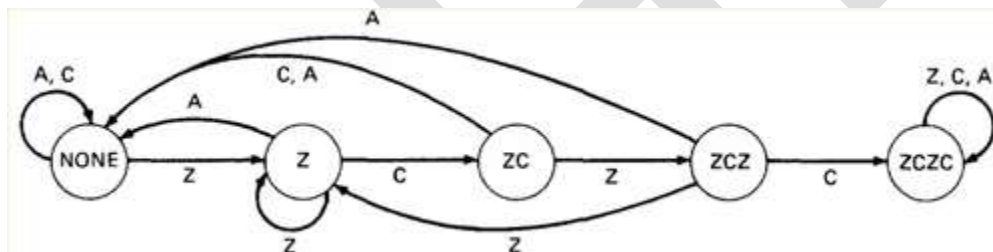


Fig 5.8. One-Time ZCZC Sequence-Detector State Graph.

2. Good State Graphs And Bad

2.1. General

some principles for judging:

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 specified 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. **

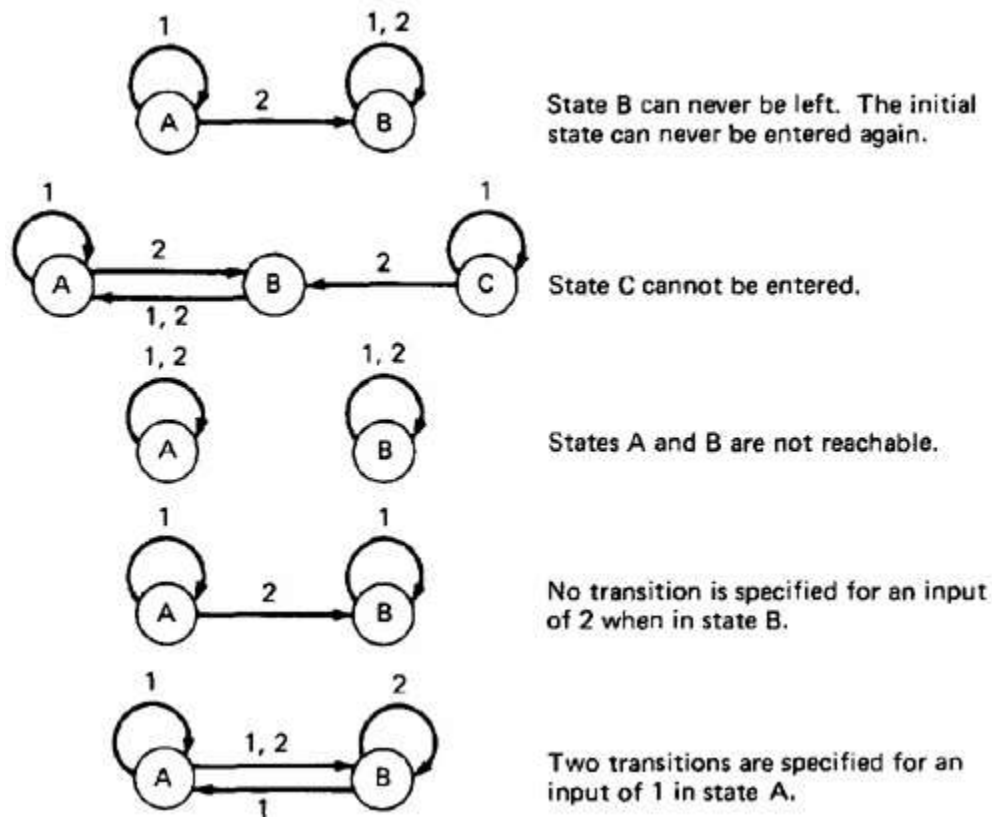


Fig 5.9. Improper State Graphs.

2.2. State Bugs

2.2.1. Number of States

The number of states in a state graph is the number of states we choose to recognize or model. Because explicit state-table mechanization is not typical, the opportunities for missing states abound. Find the number of states as follows:

1. Identify all the component factors 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.

2.2.2. Impossible States

Because the states we deal with inside computers are not the states of the real world but rather a numerical representation of real-world states, the “impossible” states can occur. A robust piece of software will not ignore impossible states but will recognize them and invoke an illogical-condition handler when they appear to have occurred. That handler will do whatever is necessary to reestablish the system’s correspondence to the world.*

2.2.3. Equivalent States

Two states are **equivalent** if every sequence of inputs starting from one state produces exactly the same sequence of outputs when started from the other state.

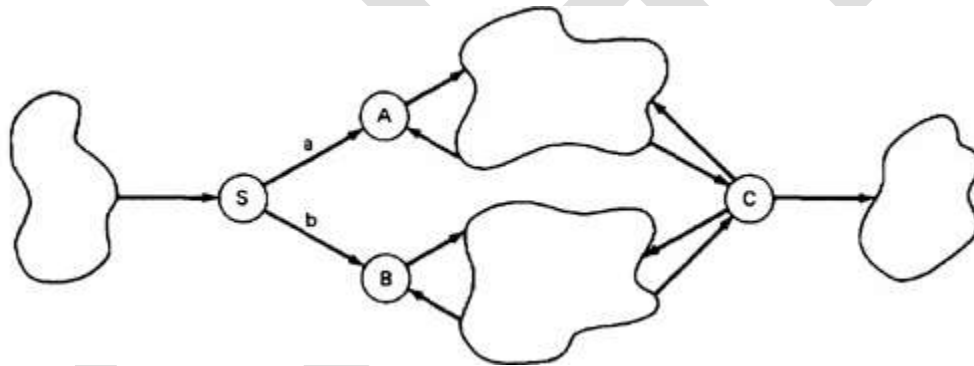


Fig 5.10. Equivalent States.

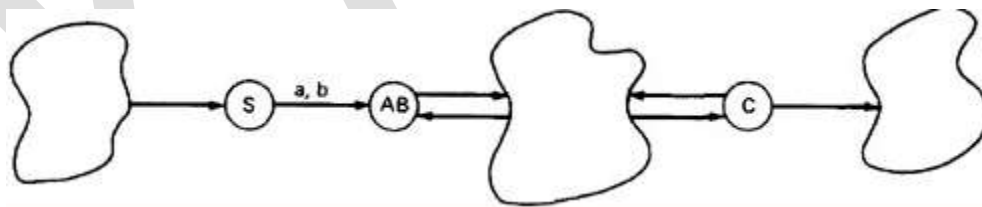


Fig 5.11 Equivalent States of Figure 5.10 Merged.

Equivalent states can be recognized by the following procedures:

1. The rows corresponding to the two states are identical with respect to input/output/next state but the name of the next state could differ. The two states are

differentiated only by the input that distinguishes between them.

2. There are two sets of rows which, except for the state names, have identical state graphs with respect to transitions and outputs. The two sets can be merged

2.3. Transition Bugs

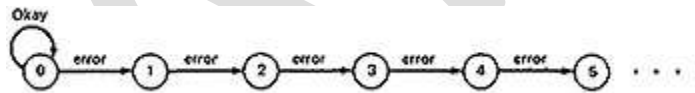
2.3.1. Unspecified and Contradictory Transitions

Every input-state combination must have a specified transition. If the transition is impossible, then there must be a mechanism that prevents that input from occurring in that state. Exactly one transition must be specified for every combination of input and stat. Ambiguities are impossible because the program will do something (right or wrong) for every input

2.3.2. An Example

Specifications are one of the most common sources of ambiguities and contradictions. The following example illustrates how to convert a specification into a state graph and how contradictions can come about. Here is the first statement of the specification:

Rule 1: The program will maintain an error counter, which will be incremented whenever there's an error. The initial state graph might look like this:



There are only two input values, “okay” and “error.” A state table will be easier to work with, and it’s much easier to spot ambiguities and contradictions. Here’s the first state table:

INPUT

STATE	OKAY	ERROR
0	0/none	1/
1		2/
2		3/
3		4/

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS: III BSC CT

COURSE NAME: SOFTWARE TESTING

COURSE CODE: 17CTU501B

UNIT: V (Logic-Based Testing and State Graphs) BATCH-2016-2019

4		5/
5		6/
6		7/
7		8/

There are no contradictions yet, but lots of ambiguities

Here are the rest of the rules; study them to see if you can find the problems, if any:

Rule 2: If there is an error, rewrite the block.

Rule 3: If there have been three successive errors, erase 10 centimeters of tape and then rewrite the block.

Rule 4: If there have been three successive erasures and another error occurs, put the unit out of service.

Rule 5: If the erasure was successful, return to the normal state and clear the error counter.

Rule 6: If the rewrite was unsuccessful, increment the error counter, advance the state, and try another rewrite.

Rule 7: If the rewrite was successful, decrement the error counter and return to the previous state.

Adding rule 2, we get

INPUT

STATE	OKAY	ERROR
0	0/NONE	1/REWRITE
1		2/REWRITE
2		3/REWRITE
3		4/REWRITE

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS: III BSC CT

COURSE NAME: SOFTWARE TESTING

COURSE CODE: 17CTU501B

UNIT: V (Logic-Based Testing and State Graphs) BATCH-2016-2019

4		5/REWRITE
5		6/REWRITE
6		7/REWRITE
7		8/REWRITE

Rule 3: If there have been three successive errors, erase 10 centimeters of tape and then rewrite the block.

INPUT

STATE	OKAY	ERROR
0	0/NONE	1/REWRITE
1		2/REWRITE
2		3/REWRITE, ERASE, REWRITE
3		4/REWRITE, ERASE, REWRITE
4		5/REWRITE, ERASE, REWRITE
5		6/REWRITE, ERASE, REWRITE
6		7/REWRITE, ERASE, REWRITE
7		8/REWRITE, ERASE, REWRITE

Rule 3, if followed blindly, causes an unnecessary rewrite. It's a minor bug, so let it go for now, but it pays to check such things. There might be an arcane security reason for rewriting, erasing, and then rewriting again.

Rule 4: If there have been three successive erasures and another error occurs, put the unit out of service.

INPUT

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS: III BSC CT

COURSE NAME: SOFTWARE TESTING

COURSE CODE: 17CTU501B

UNIT: V (Logic-Based Testing and State Graphs) BATCH-2016-2019

STATE	OKAY	ERROR
0	0/NONE	1/RW
1		2/RW
2		3/ER, RW
3		4/ER, RW
4		5/ER, RW
5		6/OUT
6		
7		

Rule 4 terminates our interest in this state graph so we can dispose of states beyond 6. The details of state 6 will not be covered by this specification; presumably there is a way to get back to state 0. Also, we can credit the specifier with enough intelligence not to have expected a useless rewrite and erase prior to going out of service.

Rule 5: If the erasure was successful, return to the normal state and clear the counter.

INPUT

STATE	OKAY	ERROR
0	0/NONE	1/RW
1		2/RW
2		3/ER, RW
3	0/NONE	4/ER, RW
4	0/NONE	5/ER, RW

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS: III BSC CT

COURSE NAME: SOFTWARE TESTING

COURSE CODE: 17CTU501B

UNIT: V (Logic-Based Testing and State Graphs) BATCH-2016-2019

5	0/NONE	6/OUT
6		

Rule 6: If the rewrite was unsuccessful, increment the error counter, advance the state, and try another rewrite.

Because the value of the error counter is the state, and because rules 1 and 2 specified the same action, there seems to be no point to rule 6 unless yet another rewrite was wanted. Furthermore, the order of the actions is wrong. If the state is advanced before the rewrite, we could end up in the wrong state. The proper order should have been: output = attempt-rewrite and then increment the error counter.

Rule 7: If the rewrite was successful, decrement the error counter and return to the previous state.

INPUT

STATE	OKAY	ERROR
0	0/NONE	1/RW
1	0/NONE	2/RW
2	1/NONE	3/ER, RW
3	0/NONE 2/NONE	4/ER, RW
4	0/NONE 3/NONE	5/ER, RW
5	0/NONE 4/NONE	6/OUT
6		

Rule 7 got rid of the ambiguities but created contradictions. The specifier's intention was

probably:

Rule 7A: If there have been no erasures and the rewrite is successful, return to the previous state.

The only thing you can assume is that it's unlikely that a satisfactory implementation will result from a contradictory specification

2.3.3. Unreachable States

An **unreachable state** is like unreachable code—a state that no input sequence can reach. Unreachable states can come about from previously “impossible” states. There are two possibilities: (1) There is a bug; that is, some transitions are missing. (2) The transitions are there, but you don't know about it; in other words, there are other inputs and associated transitions to reckon with.

2.3.4. Dead States

A **dead state**, (or set of dead states) is a state that once entered cannot be left. 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. The initialization states are unreachable to the working states, and the working states are dead to the initialization states.

2.4. Output Errors

Output actions must be verified independently of states and transitions. The likeliest reason for an incorrect output is an incorrect call to the routine that executes the output. This is usually a localized and minor bug.

2.5. Encoding Bugs

Make it a point not to use the programmer's state numbers and/or input codes. The behavior of a finite-state machine is invariant under all encodings. That is, say that the states are numbered 1 to n. If you renumber the states by an arbitrary permutation, the finite-state machine is unchanged—similarly for input and output codes.

3. State Testing

3.1. Impact of Bugs

Let's say that a routine is specified as a state graph that has been verified as correct in all details. Program code or tables or a combination of both must still be implemented. A bug can manifest itself as one or more of the following symptoms:

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

3.2. Principles

Even though most state testing can be done as a single case in a grand tour, it's impractical to do it that way for several reasons:

1. In the early phases of testing, you'll never complete the grand tour because of bugs.
2. Later, in maintenance, testing objectives are understood, and only a few of the states and transitions have to be retested. A grand tour is a waste of time.
3. There's so much history in a long test sequence and so much has happened that verification is difficult.

The starting point of state testing is:

1. Define a set of covering input sequences 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, consists of three sets of sequences:

1. Input sequences.

2. Corresponding transitions or next-state names.
3. Output sequences.

3.3. Limitations and Extensions

-transition coverage in a state-graph model does not guarantee complete testing

Work continues and progress in the form of semiautomatic test tools and effective methods are sure to come. Meanwhile, we have the following experience:

1. Simply identifying the factors that contribute to the state, calculating the total number of states, and comparing this number to the designer's notion catches some bugs.
2. Insisting on a justification for all supposedly dead, unreachable, and impossible states and transitions catches a 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 essential that means be provided (e.g., instrumentation software) to record the sequence of states (e.g., transitions) resulting from the input sequence and not just the outputs that result from the input sequence.

3.4. What to Model

Because every combination of hardware and software can in principle be modeled by a sufficiently complicated state graph, this representation of software behavior is applicable to every program.

Here are some situations in which state testing may prove useful:

1. Any processing where the output is based on the occurrence of one or more sequences of events, such as detection of specified input sequences, sequential format validation, parsing, and other situations in which the order of inputs is important.
2. Most protocols between systems, between humans and machines, between

components of a system.

3. Device drivers such as for tapes and discs that have complicated retry and recovery procedures if the action depends on the state.

4. Transactions flow where the transactions are such that they can stay in the system indefinitely—for example, online users, and tasks in a multitasking system.

5. High-level control functions within an operating system. Transitions between user states, supervisor's states, and so on. Security handling of records, permission for read/write/modify privileges, priority interrupts and transitions between interrupt states and levels, recovery issues and the safety state of records and/or processes with respect to recording recovery data.

6. The behavior of the system with respect to resource management and what it will do when various levels of resource utilization are reached. Any control function that involves responses to thresholds where the system's action depends not just on the threshold value, but also on the direction in which the threshold is crossed. This is a normal approach to control functions. A threshold passage in one direction stimulates a recovery function, but that recovery function is not suspended until a second, lower threshold is passed going the other way.

7. A set of menus and ways that one can go from one to the other. The currently active menus are the states, the input alphabet is the choices one can make, and the transitions are invocations of the next menu in a menu tree. Many menu-driven software packages suffer from dead states—menus from which the only way out is to reboot.

8. Whenever a feature is directly and explicitly implemented as one or more state-transition tables.

3.5. Getting the Data

getting the data on which the model is to be based is half the job or more. There's no magic for doing that: reading documents, interviews, and all the rest

3.6. Tools

The telecommunications industry, especially in telephony, has been using finite-state machine implementations of control functions for decades (BAUE79). They also use several languages/systems to code state tables directly. Similarly, there are tools to do the same for hardware logic designs. The bad news is that these systems and languages are proprietary, of the home-brew variety, internal, and/or not applicable to the general use of software implementations of finite-state machines.



KARPAGAM ACADEMY OF HIGHER EDUCATION

DEPARTMENT OF COMPUTER SCIENCE

III B.Sc IT (Batch 2017-2020)

SOFTWARE TESTING (17ITU501B)

PART - A OBJECTIVE TYPE/MULTIPLE CHOICE QUESTIONS

ONLINE EXAMINATIONS

ONE MARKS QUESTIONS

UNIT 1

S.NO	Question	Option 1	Option 2	Option 3	Option 4	Answer
1	_____ is the number of delimiters appearing at a field boundary may be variable	Backus-Naur form	Back_Naur form	Backus_Nour form	None of these	Backus_Nour form
2	The _____ may be optional or several alternate formats may be acceptable	Declaration	Transaction	Method	Syntax	Syntax
3	Field-value errors are clearly a _____ issue	Data declaration	Data Abstraction	Data Validation	Data Transaction	Data Validation
4	A scripting language and processor such as _____ has the features needed to automate the replacement of good substrings by bad ones on the fly	Flow of characters	Array of characters	String of characters	None of these	String of characters
5	_____ is also an excellent way of convincing a novice tester that testing is infinite and that the tester's problem is not generating tests but knowing which ones to cull	Data declaration	Data Abstraction	Data Validation	Data Transaction	Data Validation
6	Build or buy a _____ that the program automatically sequences through a set of test cases usually stored as data	String generator	String recognizer	String of characters	Syntax	String generator
7	Which of the following is under compilation steps:	Strings	Delimiters	Literals	None of these	Delimiters
8	_____ kind of error causes adjacent fields to merge	Missing delimiters	Delimiters	Wrong delimiters	None of these	Missing delimiters

9	_____ are characters or strings placed between two fields to denote where one ends and the other begin.	Missing delimiters	Delimiters	Wrong delimiters	None of these	Wrong delimiters
10	The tester attempts to generate strings and is said to be a _____	Missing delimiters	Too many delimiters	Wrong delimiters	delimiters	Too many delimiters
11	_____ routine is designed to recognize strings that have been	Missing delimiters	Too many delimiters	Tolerant Delimiters	Wrong delimiters	Tolerant Delimiters
12	Every input can be considered as if it were a _____	Path testing	domain-testing	validate testing	Segment testing	domain-testing
13	_____ consists (in part) of checking the input for correct syntax	transaction	Declaration	Syntax	None of these	transaction
14	Every input has _____	complete	consistent	neither consistent nor complete	None of these	neither consistent nor complete
15	Internal and external inputs conform to formats, which can usually be expressed in _____	capture	capture/replay	replay	None of these	capture/replay
16	_____ several different delimiters are used and there are rules that specify which can be used where.	Driver	Hardware	software	None of these	Driver
17	The string or field value that may be acceptable at one instant may not be acceptable at the next because validity depends on the _____	CACL	CALC	CASL	None of these	CASL
18	If you get the designer to create the first version of the BNF specification, you may find that it is _____	Path testing	black box testing	white box testing	syntax testing	syntax testing
19	A _____ system captures your keystrokes and stuff sent to the screen and stores them for later execution	2	3	4	5	3
20	Compilation consists of _____ main steps	Lexical analysis	Parsing	code production	All of the above	All of the above

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021
Department of CS,CA & IT

Fifth Semester
FIRST INTERNAL EXAMINATION - July 2019
Software testing

Class & Section: III B.Sc (CT)
Date & Session : 22.7.18FN
Subj.Code: 17CTU501B

Duration: 2 hours
Maximum marks: 50 marks

-
1. Black box testing is sometime referred as _____
a) Glass **b) Functional** c) Structural d) All the above
 2. White box testing is sometime called as _____.
a) **Glass** b) Functional c) Structural d) All the above
 3. The term ANSI is referred _____
a) American Networking Standard Institute
b) American National Standard Institute
c) American National Standardize Institute
d) American National Standards Institute
 4. Usage of equal partitioning is _____ test cases
a) To Reduce b) To Increase c) To Avoid d) To Allow
 5. Other name for equal partitioning _____
a) **Equivalence Classes** b) Equivalence Objects
c) Equivalence Methods d) Equivalence Directory
 6. State testing is performed for _____
a) Verification Of Programming Codes **b) Verification Of Programming Logic**
c) Transmission Of Programming Codes d) Transmission Of Programming Logic
 7. Static white box testing is the process of examining _____
a) Design b) Code c) Logic **d) Both A & B**
 8. View in software design , architecture or code for bugs is called _____
a) Logic Analysis **b) Structural Analysis** c) Design Analysis d) Code Analysis
 9. Bottom – up right own modules are called _____
a) Drivers b) Dataflow c) Big-Bang **d) Test Drivers**
 - 10) Top-down sometime called as _____
a) Drivers b) Dataflow **c) Big-Bang** d) Test Drivers
 - 11) A process block is a sequence of program statements uninterrupted by
a) **Decisions** b) Process Block c) Case Statement d) Instruction
 12. A _____ is a multi-way branch or decisions
a) Decisions b) Process Block **c) Case Statement** d) Instruction
 13. _____ Execute all possible control flow paths through the program

Two Approaches to testing

Test-to-pass: apply simple and straight forward test cases.

- Test-to-fail : intend to find bugs by any means

23. What is static white box testing?

ANSWER:

Static white box testing

It is the process of examining design and code. It includes

- Reviewing software design, architecture or code for bugs with execution.
- Also called structured analysis
- This method finds bugs early during the development process
- It finds bugs that are difficult to uncover

PART-B

[3 * 8 = 24 Marks]

Answer any 10 Questions

24. a) Describe in detail about the strategies used to perform high-level review of specification.

ANSWER:

Performing High Level Review of Specification

- While testing don't jump straight and look for bugs in code.
- Stand back and view it from high level.
- If there is better understanding of why and how then examination will be in detail.

1. Pretend to be a customer:

- Get known about end user.
- Understand customer expectation.
- Don't assume anything to be correct.
- If bugs are found, it is better.
- Test security of the software also.

2. Research existing standards and guidelines

- Back in days every software structure of every company like Microsoft and Apple are different. So it requires retraining.
- Now all software and hardware are standardized. So products are similar in look and feel.

- Standard should be strictly adhered
- Specifying Guidelines is optional but should be followed.

Example of standards and guidelines:

- Corporate terminology and conventions.
- Industry requirements.
- Government standards.
- Graphical User Interface
- Security standards.

Tester – It defines guidelines and standards applied to product developed.
A tester tests if standards are used and not overlooked.

3. Review and test similar software

The following are the things to be looked when reviewing competitive product.

- Scale – features included
- Complexity
- Testability
- Quality / Reliability
- Security

Read online and printed software reviews and articles about competitor.

(or)

b) Explain in detail about low level specification test techniques.

ANSWER:

1. Low Level Specification Test technique

- Testing specification at lower level.

Specification attributes checklists – The following attributes must be verified

1. Complete. Is anything missing or forgotten? Is it thorough? Does it include everything necessary to make it stand alone?
2. Accurate. Is the proposed solution correct? Does it properly define the goal? Are there any errors?
3. Precise, Unambiguous, and Clear. Is the description exact and not vague? Is there a single interpretation? Is it easy to read and understand?
4. Consistent. Is the description of the feature written so that it doesn't conflict with itself or other items in the specification?
5. Relevant. Is the statement necessary to specify the feature? Is it extra information that should be left out? Is the feature traceable to an original customer need?
6. Feasible. Can the feature be implemented with the available personnel, tools, and resources within the specified budget and schedule?
7. Code-free. Does the specification stick with defining the product and not the underlying software design, architecture, and code?

8. Testable. Can the feature be tested? Is enough information provided that a tester could create tests to verify its operation?

Specification Terminology Characteristics

- Always, Every, All, None, Never. If these words are seen such as these that denote something as certain or absolute, make sure that it is, indeed, certain.
- Certainly, Therefore, Clearly, Obviously, Evidently. These words tend to persuade you into accepting something as a given. Don't fall into the trap.
- Some, Sometimes, Often, Usually, Ordinarily, Customarily, Most, Mostly. These words are too vague. It's impossible to test a feature that operates "sometimes."
- Etc., And So Forth, And So On, Such As. Lists that finish with words such as these aren't testable. Lists need to be absolute or explained so that there's no confusion as to how the series is generated and what appears next in the list.
- Good, Fast, Cheap, Efficient, Small, Stable. These are unquantifiable terms. They aren't testable. If they appear in a specification, they must be further defined to explain exactly what they mean.
- Handled, Processed, Rejected, Skipped, Eliminated. These terms can hide large amounts of functionality that need to be specified.
- If...Then (but missing Else). Look for statements that have "If...Then" clauses but don't have a matching "Else." Ask yourself what will happen if the "if" doesn't happen.

- 25) a) Write in detail about the approach used in equivalence partitioning method. Provide neat sketches wherever necessary.

ANSWER:

. Equivalence partitioning (or) Equivalence classing

- It is means by which test cases are selected.
- Process of reducing huge set of possible test cases into smaller ones. But equally effective.

Example : - calculator

- Not possible to check all cases of adding 2 numbers together.
- Check 1+1, 1+2, 1+3, safely assure if 1+5, 1+6 also works correct.

Example :

1. 1+999999999999 looks different and so may have a bug in it.

2. We provide five options to copy and paste. But all options perform same operation

- a) Click copy
- b) type c or C if menu displayed
- c) ctrl + C or ctrl + shift + C
- d) Click command to menu
- e) press Ctrl + C



Figure 2.5 Multiple ways to invoke the copy function with same result.

3. Giving name in **Save As** dialog box - A name must be checked for a valid character, invalid character, valid length, name too short and name too long.



Figure 2.6 File Name text box in the Save As dialog box illustrates several equivalence partition possibilities.

A Windows filename can contain any characters except \ / : * ? " < > and |. Filenames can have from 1 to 255 characters. If test cases are created for filenames, have equivalence partitions for valid characters, invalid characters, valid length names, names that are too short, and names that are too long.

Goals of Equivalence partitioning

- The aim of equivalence partitioning should not be to reduce number of test cases
- This process may lead to bugs.
- If the person is new to testing then get classes from an experienced person.

(or)

b) What strategies are followed to perform the data testing? With an example program describe its process and uses.

ANSWER:

. Data Testing

- Divide software into data and program
- Data – input, output, printout, mouse clicks etc.,
- Program – flow, transitions, logic, computation.

Examples of data

- Words typed in word processor.
- Numbers typed in spreadsheet
- Number of shots in your game
- Picture printed by your software
- Backup files stored on floppy disk
- Data sent to modem over phone lines.

Tester should reduce test cases by Equivalence partitioning based on few concepts. They are boundary conditions, sub-boundary conditions, nulls and bad data.

1. Boundary conditions

- If it is possible to walk along the edge of a cliff, then it also possible to walk on the middle.
- If software operates on edge of its capabilities, almost operates under normal condition.



Figure 2.7 Software boundary is much like the edge of a cliff.

Basic program

- 1) Rem create a 10 element integer array
- 2) Rem initialize each element to -1
- 3) Dim data(10) as integer
- 4) Dim i as integer
- 5) For i = 1 to 10
- 6) Data(i) = -1
- 7) Next i
- 8) End

- This program actually creates a data array of 11 elements from data (0) to data (10).
- The program loops from 1 to 10 and initializes those values of the array to 1, but since the first element of our array is data (0), it doesn't get initialized.
- When the program completes, the array values look like this:

data(0) = 0	data(6) = 1
data(1) = 1	data(7) = 1
data(2) = 1	data(8) = 1
data(3) = 1	data(9) = 1
data(4) = 1	data(10) = 1
data(5) = 1	

- The data (0)'s value is 0, not 1.
- If the same programmer later forgot about, or a different programmer wasn't aware of how this data array was initialized, he might use the first element of the array, data (0), thinking it was set to 1.
- Problems such as this are very common and, in large complex software, can result in very nasty bugs.

Types of Boundary Conditions

Boundary conditions are situations at edge of planned operational limits of free software.

When a tester is presented with a software test problem that involves identifying boundaries, he must look for the following types:

Numeric	Speed
Character	Location
Position	Size
Quantity	

And, the following are the characteristics of those types:

First/Last	Min/Max
Start/Finish	Over/Under
Empty/Full	Shortest/Longest
Slowest/Fastest	Soonest/Latest
Largest/Smallest	Highest/Lowest
Next-To/Farthest-From	

Testing the Boundary Edges

- Create 2 Equivalence partitions
- First partition should be such that it is the last value of a data and 2 points to be chosen inside boundary.
- Second partition should be chosen as the data that can cause error and 2 invalid points outside boundary

Testing outside boundary

- First -1 / last + 1
- Start -1 / finish + 1
- Less than empty / more than full
- Even slower / even faster
- Largest+1 / smallest -1

Testing

- Text allowing 1-255 character. Test by entering 1 and 255
- Flight simulator – try at ground level and maximum height allowed
- If software allows 9 digit zip code enter and test 000000000 and 999999999

Sub Boundary Condition

- Not important for end user but must be checked.
- It is also called internal boundary conditions.
Example: ASCII codes; powers-of-two
- These conditions are discussed with programmers and checked.
- It checks for default, empty, blank, null, zero and more.
- It also checks data that are invalid, wrong, incorrect and garbage data
- It includes testing the logic flow of software

26) a) Explain about Generic Code Review Checklist

ANSWER:

Generic Code Review Checklist

There are many types of errors that must be listed before going into the process of testing.
They are

1. Data reference error

- Is uninitialized variable referenced?
- Is array out of bound
- Is there any problem in index references of array?
- Is any variable used instead of constant?
- Is floating point number assigned to integer variable?
- Is memory allocated for pointers?

2. Data declaration error

- Is all variables assigned correct length, type etc?
- Check if variable initialized while declared
- Is there variable name with similar name?
- Are there any unreferenced variable
- Are all variable explicitly declared?

3. Computation error

- Existence of two variables of different data type – integer and float
- Existence of two variable of different length – byte and word
- Variable in assignment smaller than Right hand side of expression
- Overflow, underflow
- Division by zero
- Value of variable outside range. Example percentage value only between 0-100
- Confusion in order of expression if there are multiple operators

4. Comparison error

- Is comparison correct?
- Is there comparison between fractional and floating point?
- Is there any confusion in order of evaluation in Boolean expression?
- Whether operators in Boolean expression are Boolean?

5. Control flow error

- Matching groups, begin-end, do-while
- Whether loop terminates correctly?
- Possibility if a loop never executes
- Switch index exceeds number of existing branches
- Unexpected flow through loop

6. Subroutine parameter error

- Type, size and order of precedence correct or not
- Multiple entry point in subroutine
- Change in order of parameters if send as constant
- Is there any alteration in parameters?
- Whether formal and actual arguments match?
- Whether definition of global variable same everywhere

7. I/O error

- Data format read/printed
- Device not ready
- Device disconnected
- Error handled in expected way
- Check error message for spelling and grammar

8. Other checks

- Will the software work with language other than English?
- Will the software work with other compilers and CPU's?
- Will the software work with different amount of memory, hardware, sound card etc?
- Whether compilation of program produce warning or informational message

(or)

b)Discuss about the Formal review

ANSWER:

Formal review

- It is a meeting between two programmers
- It is a rigorous method of inspection of software design and code.

4 elements:

1. Identify problem - find wrong and missing items. The criticism is only to the product. It should not extend to the programmer
2. Follow rules - amount of code to be reviewed, time spent, what can be commented on etc. are provided in the rules
3. Prepare for review – the reviewers must know what is their duty, role, responsibility during the review and fulfill them.
4. Write a report: Finally after the review a written report summarizing the result of review must be prepared.

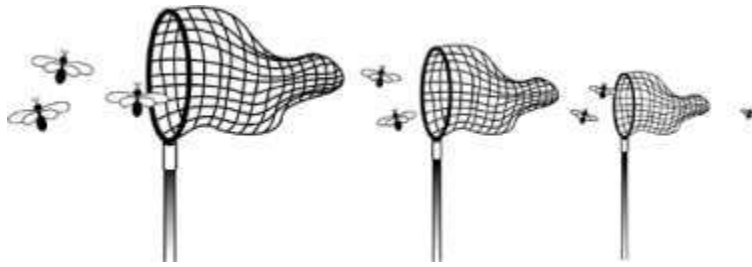


Figure 2.12 Formal reviews are the first nets used in catching bugs

Use of formal review

- It increases communication between testers
- Improves quality of the software product
- Team camaraderie
- It provides a solution even for tough problem.

1. Peer review or buddy review

- It is done by designer, programmer and tester together.
- They review the code and look for problems
- Since it is informal method of testing, the testers may not follow the four elements of testing

2. Walkthroughs

- Programmer who wrote the code formally presents the application to a small group of reviewers
- One senior programmer must be a reviewer
- The reviewer can write comments and questions
- There will be large number of people. So this method looks much formal
- Presenter writes report on how bugs were found and how to solve it

3. Inspections

- It is a highly structured method of testing
- The participants here needs training
- The presenter is not a programmer.
- Some other person learn and explain the code to others
- Other participants are called inspectors
- Using this method identify different bugs
- The review process is done backwards
- After the testing process is done, prepare a written report.
- This report identifies rework
- A re-inspection is done to locate remaining bugs