UNIT-1

INTRODUCTION TO TESTING,FLOW GRAPHS AND PATH TESTING

1.The purpose of testing

1.1 what we do

Software testing is to ensure the quality and the primary objective of testing is to find bugs. Testing verifies that the system meets the different requirements including functional, performance, reliability, security and usability. Software testing contributes to improving the quality of product

Software Testing consumes at least half of the time and work required to produce a functional program.

MYTH: Good programmers write code without bugs. (Its wrong!!!)

But in reality even well written programs still have 1-3 bugs per hundred statements

1.2 Productivity and Quality in software:

In production of consumer goods and other products, every manufacturing stage is subjected to quality control and testing from component to final stage.

If flaws(errors) are discovered at any stage, the product is either discarded or cycled back for rework and correction.

Productivity is measured by the sum of the costs of the material, the rework, and the discarded components, and the cost of quality assurance and testing.

There is a trade off between quality assurance costs and manufacturing costs: If sufficient time is not spent in quality assurance, the reject rate will be high and so will be the net cost. If inspection is good and all errors are caught as they occur, inspection costs will dominate, and again the net cost will suffer.

Testing and Quality assurance costs for 'manufactured' items can be as low as 2% in consumer products or as high as 80% in products such as space-ships, nuclear reactors, and aircrafts, where failures threaten life.

Where as the manufacturing cost of a software is trivial. The biggest part of software cost is the cost of bugs: the cost of detecting them, the cost of correcting them, the cost of designing tests that discover them, and the cost of running those tests.

For software, quality and productivity are indistinguishable because the cost of a software copy is trivial.

1.3.Goals for testing

Testing and Test Design are parts of quality assurance Testing and test design do not prevent bugs they should be able to correct should also focus on bug prevention.

Tests should provide clear diagnosis so that bugs can easily be corrected

A prevented bug is better than a detected and corrected bug.

Test then code the ideal test activity would be so successful at bug prevention that actual testing would be unnecessary

A test design must document all the expectations the test procedure in detail and the results of the actual test.

1.4.Phases In A Tester's Mental Life

Why testing: Testing is to ensure the quality of a software and to find bugs. It is characterized by 5 phases

Phase 0-Thinking:

There is no difference between testing and debugging. Phase 0 thinking was the norm in early days of software development till testing emerged as a discipline.

Phase 1-Thinking-The Software Works: The purpose of testing here is to show that software works. Highlighted during the late 1970s. This failed because the probability of showing that software works 'decreases' as testing increases. *i.e.* The more you test, the more likely you'll find a bug.

Phase 2-Thinking The Software Does Not Work: The purpose of testing is to show that software doesn't work. Phase 2 thinking leads to strong, revealing tests Phase 2 leads to never ending sequence of tests .the trouble with phase 2 thinking is that we don't know when to stop

Phase 3-Thinking-Test for Risk Reduction: The purpose of testing is not to prove anything but to reduce the perceived risk of not working to an acceptable value The product is released when the confidence on that product is high enough. (Note: This is applied to large software products with millions of code and years of use.)

Phase 4-Thinking-A state of mind: Testability is the factor considered here. One reason is to reduce the labour of testing. Other reason is to check the testable and non-testable code. Testable code has fewer bugs than the code that's hard to test. Identifying the testing techniques to test the code is the main key here.

Test Design:

We know that the software code must be designed and tested, but many appear to be unaware that tests themselves must be designed and tested. Tests should be properly designed and tested before applying it to the actual code.

The programming process should be described as “design, test design, code, test code, program inspection, test inspection, test debugging, test execution, program debugging, testing”,

Testing is'nt everything*:*

There are approaches other than testing to create better software. Methods other than testing include:

Inspection Methods: Methods like walkthroughs, deskchecking, formal inspections and code reading appear to be as effective as testing but the bugs caught donot completely overlap.

Design Style: While designing the software itself, adopting stylistic objectives such as testability, openness and clarity can do much to prevent bugs.

Static Analysis Methods: Includes formal analysis of source code during compilation. In earlier days, it is a routine job of the programmer to do that. Now, the compilers have taken over that job

Languages: The source language can help reduce certain kinds of bugs. Programmers find new bugs while using new languages.

Development Methodologies and Development Environment: The development process and the environment in which that methodology is embedded can prevent many kinds of bugs.

1.7 The pesticide paradox and the complexity barrier:

1.The Pesticide Paradox Law: Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are Ineffectual

As we progress by enhancements and correction of bugs we may say software gets better, however it may not be quite that

2.The Complexity Barrier Law: Software complexity grows to the limits of our ability to manage the complexity. Simpler bugs are corrected and the software is enhanced with new features increasing the complexity.

2. DICHOTOMIES

1.Testing Versus Debugging

TESTING	DEBUGGING
1.Testing starts with known conditions and we can predict output	1.Debugging starts with unknown initial conditions
2.Testing is used to test the code whether it has bugs or not	2.Debugging is used to rectify the error which caused by program failure
3.Testing can be planned, designed and scheduled	3.Debugging can not be planned
4.It proves programmer failure	4.It gives programmer justification
5.It can be done by outsider	5.It can be done by insider
6.Design knowledge is not necessary	6.Design knowledge is important
7.Most of the test execution is automated	7.Automated debugging is not there still it is a dream

2.Function Versus Structure

Function	Structure
1.In Functional testing the program or system is treated as a black box testing	1.Structural testing is treated as a white box testing
2. It is subjected to inputs, and its outputs are verified for conformance to specified behavior.	2.It is subjected to its program implementation
3. Functional testing takes the user point of view- bother about functionality and features and not the program's implementation	3. Structural testing does look at the implementation details. Things such as programming style, control method, source language, database design, and coding details dominate structural testing
4. Functional tets can detect all bugs but would take infinite time to do so.	4. Structural tests are inherently finite but cannot detect all errors even if completely executed.

3. Designer Versus Tester: Test designer is the person who designs the tests where as the tester is the one actually tests the code. During functional testing, the designer and tester are probably different persons. During unit testing, the tester and the programmer merge into one person. Tests designed and executed by the software designers are by nature biased towards structural consideration and therefore suffer the limitations of structural testing.

4.Modularity Versus Efficiency: A module is a discrete, well-defined, small component of a system. Smaller the modules, difficult to integrate; larger the modules, difficult to understand. Both tests and systems can be modular. Testing can and should likewise be

organised into modular components. Small, independent test cases can be designed to test independent modules

5.Small Versus Large: Programming in large means constructing programs that consists of many components written by many different programmers. Programming in the small is what we do for ourselves in the privacy of our own offices. Qualitative and Quantitative changes occur with size and so must testing methods and quality criteria.

6.Builder Versus Buyer: Most software is written and used by the same organization. Unfortunately, this situation is dishonest because it clouds accountability. If there is no separation between builder and buyer, there can be no accountability.

The different roles / users in a system include:

Builder: Who designs the system and is accountable to the buyer.
Buyer: Who pays for the system in the hope of profits from providing services.
User: Ultimate beneficiary or victim of the system. The user's interests are also guarded by.
Tester: Who is dedicated to the builder's destruction.
Operator: Who has to live with the builders' mistakes, the buyers' murky (unclear) specifications, testers' oversights and the users' complaints.

3.Model for testing

3.1 The project

Testers will do testing the interfaces in a program in different ways. A real world context characterized by the following model project

Application: It is a real time system that must provide timely responses to user requests for services

Staff: The Programming staff consists of twenty to thirty programmers and big enough to use specialists for some parts of the system design

Schedule: The project will take 24 months from the start to design to formal acceptance by the customer Specification: The specification is good, it is functionally detailed without constraining the design

Acceptance test: The system will be accepted only after a formal acceptance test

Personnel: The staff is professional and experienced in programming and in the application

Standards: Programming and testing standards and are usually followed

1.Documentation is good

2.There is an internal, semiformal quality assurance function

3.The database is centrally developed and administered

Objectives:

The system is the first of many similar systems that will be implemented in the future. No two will be identical but they will have 75% of the code in the common.

History:

One programmer will quit before his components are tested. Another programmer will be fired before testing begins, excellent work but poorly documented. Once component will have to be redone after unit testing, a superb piece of work that defines integration

Our model project is a typical well run successful project with a share of glory and catastrophe

OVERVIEW

The process starts with a program embedded in an environment, such as computer,a n operating system

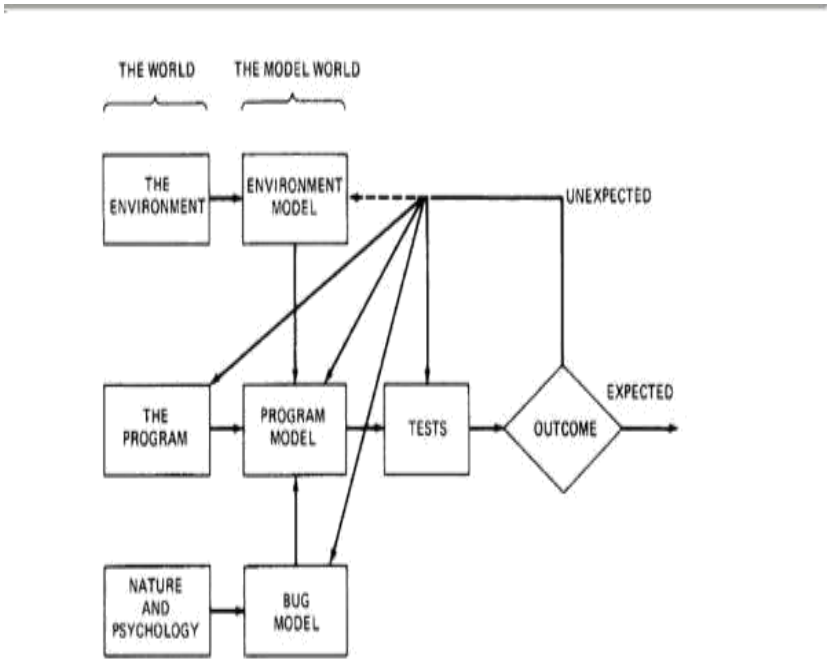


Figure 1: A model of testing

It includes three models: A model of the environment, a model of the program and a model of the expected bugs.

ENVIRONMENT:

A Program's environment is the hardware and software required to make it run. For online systems, the environment may include communication lines, other systems, terminals and operators. The environment also includes all programs that interact with and are used to create the program under test - such as OS, linkage editor, loader, compiler, utility routines. Because the hardware and firmware are stable, it is not smart to blame the environment for bugs.

PROGRAM:

Most programs are too complicated to understand in detail. The concept of the program is to be simplified inorder to test it. If simple model of the program doesnot explain the unexpected behaviour, we may have to modify that model to include more facts and details. And if that fails, we may have to modify the program.

BUGS:

Bugs are more insidious (deceiving but harmful) than ever we expect them to be.An unexpected test result may lead us to change our notion of what a bug is and our model of bugs. Some optimistic notions that many programmers or testers have about bugs are usually unable to test effectively and unable to justify the dirty tests most programs need.

The beliefs about bugs are

Benign Bug Hypothesis: The belief that bugs are nice, tame and logical. (Benign: Not Dangerous)

Bug Locality Hypothesis: The belief that a bug discovered with in a component effects only that component's behavior.

Control Bug Dominance: The belief that errors in the control structures (if, switch etc) of programs dominate the bugs.
Code / Data Separation: The belief that bugs respect the separation of code and data.

Lingua Salvator Est: The belief that the language syntax and semantics (e.g. Structured Coding, Strong typing, etc) eliminates most bugs.

Corrections Abide: The mistaken belief that a corrected bug remains corrected
Silver Bullets: The mistaken belief that X (Language, Design method, representation, environment) grants immunity from bugs.

Sadism Suffices: The common belief (especially by independent tester) that a sadistic streak, low cunning, and intuition are sufficient to eliminate most bugs. Tough bugs need methodology and techniques

Angelic Testers: The belief that testers are better at test design than programmers are at code design.

TESTS:

Tests are formal procedures, Inputs must be prepared, Outcomes should predicted, tests should be documented, commands need to be executed, and results are to be observed. *All these errors are subjected to error* We do three distinct kinds of testing on a typical software system. They are:

Unit / Component Testing: A **Unit** is the smallest testable piece of software that can be compiled, assembled, linked, loaded etc. A unit is usually the work of one programmer and consists of several hundred or fewer lines of code. **Unit Testing** is the testing we do to show that the unit does not satisfy its functional specification or that its implementation structure

does not match the intended design structure. A **Component** is an integrated aggregate of one or more units. **Component Testing** is the testing we do to show that the component does not satisfy its functional specification or that its implementation structure does not match the intended design structure.
Integration Testing: Integration is the process by which components are aggregated to create larger components. **Integration Testing** is testing done to show that even though the components were individually satisfactory (after passing component testing), checks the combination of components are incorrect or inconsistent.

System Testing: A **System** is a big component. **System Testing** is aimed at revealing bugs that cannot be attributed to components. It includes testing for performance, security, accountability, configuration sensitivity, startup and recovery.
Role of Models: The art of testing consists of creating , selecting, exploring, and revising models. Our ability to go through this process depends on the number of different models we have at hand and their ability to express a program's behavior.

4.THE CONSEQUENCES OF BUGS

IMPORTANCE OF BUGS:

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

Frequency: How often does that kind of bug occur? Pay more attention to the more frequent bug types

Correction Cost: What does it cost to correct the bug after it is found? The cost is the sum of 2 factors: (1) the cost of discovery (2) the cost of correction. These costs go up dramatically later in the development cycle when the bug is discovered. Correction cost also depends on system size.

Installation Cost: Installation cost depends on the number of installations: small for a single user program but more for distributed systems. Fixing one bug and distributing the fix could exceed the entire system's development cost.

Consequences: What are the consequences of the bug? Bug consequences can range from mild to catastrophic.

A reasonable metric for bug importance is

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

How Bugs Affects us-Consequences: The consequences of a bug can be measure in terms of human rather than machine.

Some consequences of a bug on a scale of one to ten are:

Mild: The symptoms of the bug offend us aesthetically (gently); a misspelled output or a misaligned printout.

Moderate: Outputs are misleading or redundant. The bug impacts the system's performance.

Annoying: The system's behaviour because of the bug is dehumanizing. *E.g.* Names are truncated orarbitarily modified.

Disturbing: It refuses to handle legitimate (authorized / legal) transactions. The ATM wont give you money. My credit card is declared invalid.

Serious: It loses track of its transactions. Not just the transaction itself but the fact that the transaction occurred. Accountability is lost.

Very Serious: The bug causes the system to do the wrong transactions. Instead of losing your paycheck, the system credits it to another account or converts deposits to withdrawals.

Extreme: The problems aren't limited to a few users or to few transaction types. They are frequent and arbitrary instead of sporadic infrequent) or for unusual cases.

Intolerable: Long term unrecoverable corruption of the database occurs and the corruption is not easily discovered. Serious consideration is given to shutting the system down.

Catastrophic: The decision to shut down is taken out of our hands because the system fails.

Infectious: What can be worse than a failed system? One that corrupt other systems even though it doesnot fall in itself ; that erodes the social physical environment; that melts nuclear reactors and starts war.

Flexible Severity Rather Than Absolutes:

Quality can be measured as a combination of factors, of which number of bugs and their severity is only one component. Many organizations have designed and used satisfactory, quantitative, quality metrics. Because bugs and their symptoms play a significant role in such metrics, as testing progresses, you see the quality rise to a reasonable value which is deemed to be safe to ship the product.

The factors involved in bug severity are:

Correction Cost: Not so important because catastrophic bugs may be corrected easier and small bugs may take major time to debug.

Context and Application Dependency: Severity depends on the context and the application in which it is used.

Creating Culture Dependency: Whats important depends on the creators of software and their cultural aspirations. Test tool vendors are more sensitive about bugs in their software then games software vendors.

User Culture Dependency: Severity also depends on user culture. Naive users of PC software go crazy over bugs where as pros (experts) may just ignore.

The software development phase: Severity depends on development phase. Any bugs gets more severe as it gets closer to field use and more severe the longer it has been around.

THE NIGHTMARE LIST AND WHEN TO STOP TESTING

Quantifying the Nightmare :

1. List your worst software nightmares. State them in terms of the symptoms they produce and how your user will react to those symptoms.

2. Convert the consequences of each nightmare into a cost. Usually, this is a labor cost for correcting the nightmare.

3. Order the list from the costliest to the cheapest and then discard the low-concern nightmares with which you can live.

4. Measure the kinds of bugs that are likely to create the symptoms expressed by each nightmare.

5. For each nightmare, then, you've developed a list of possible causative bugs. Order that list by decreasing probability. Judge the probability based on your own bug statistics, intuition, experience, etc. The same bug type will appear in different nightmares. The importance of a bug type is calculated by multiplying the expected cost of the nightmare by the probability of the bug and summing across all nightmares:

$$\text{importance of bug type } i = \sum_{\text{all nightmares}} C_{ij} P_{(\text{bug type } i \text{ in nightmare } j)}$$

6. Rank the bug types in order of decreasing importance to you.

7. Design tests (based on your knowledge of test techniques) and design your quality assurance inspection process by using the methods that are most effective against the most important bugs.

8. If a test is passed, then some nightmares or parts of them go away. If a test is failed, then a nightmares possible, but upon correcting the bug, it too goes away. Testing, then, gives you information you can use to revise your estimated nightmare probabilities.

9. Stop testing when the probability of all nightmares has been shown to be inconsequential as a result of hard evidence produced by testing.

General : There is no universally correct way to categorize bugs. This taxonomy is not rigid. Bugs are difficult to categorize. A given bug can be put into one or another category depending on its history and the programmer's state of mind.

5. A TAXANOMY OF BUGS

There is no universally correct way categorize bugs. The taxonomy is not rigid. A given bug can be put into one or another category depending on its history and the programmer's state of mind.

The major categories are: (1) Requirements, Features and Functionality Bugs (2) Structural Bugs (3) Data Bugs (4) Coding Bugs (5) Interface, Integration and System Bugs (6) Test and Test Design Bugs.

REQUIREMENTS, FEATURES AND FUNCTIONALITY BUGS: Various categories in

Requirements, Features and Functionlity bugs include:

Requirements and Specifications Bugs:

Requirements and specifications developed from them can be incomplete ambiguous, or self-contradictory. They can be misunderstood or impossible to understand.

The specifications that don't have flaws in them may change while the design is in progress. The features are added, modified and deleted. Requirements, especially, as expressed in specifications are a major source of expensive bugs. The range is from a few percentage to more than 50%, depending on the application and environment. What hurts most about the bugs is that they are the earliest to invade the system and the last to leave.

Feature Bugs:

Specification problems usually create corresponding feature problems. A feature can be wrong, missing, or superfluous (serving no useful purpose). A missing feature or case is easier to detect and correct. A wrong feature could have deep design implications. Removing the features might complicate the software, consume more resources, and foster more bugs.

Feature Interaction Bugs:

Providing correct, clear, implementable and testable feature specifications is not enough. Features usually come in groups or related features. The features of each group and the interaction of features with in the group are usually well tested. The problem is unpredictable interactions between feature groups or even between individual features. For example, your telephone is provided with call holding and call forwarding. The interactions between these two features may have bugs. Every application has its peculiar set of features and a much bigger set of unspecified feature interaction potentials and therefore result in feature interaction bugs. Most feature bugs are rooted in human to human communication problems. One solution is to use high-level, formal specification languages or systems.

Such languages and systems provide short term support but in the long run, does not solve the problem.

Short term Support: Specification languages facilitate formalization of requirements and inconsistency and ambiguity analysis.

Long term Support: Assume that we have a great specification language and that can be used to create unambiguous, complete specifications with unambiguous complete testsand consistent test criteria.

The specification problem has been shifted to a higher level but not eliminated.

Testing Techniques for functional bugs: Most functional test techniques- that is those techniques which are based on a behavioral description of software, such as transaction flow testing, syntax testing, domain testing, logic testing and state testing are useful in testing functional bugs

STRUCTURAL BUGS:

Various categories in Structural bugs include:

Control and Sequence Bugs:

Control and sequence bugs include paths left out, unreachable code, improper nesting of loops, loop-back or loop termination criteria incorrect, missing process steps, duplicated processing, unnecessary processing, rampaging, GOTO's, ill-conceived (not properly planned) switches, sphagetti code, and worst of all, pachinko code. One reason for control flow bugs is that this area is amenable (supportive) to theoritical treatment. Most of the control flow bugs are easily tested and caught in unit testing. Another reason for control flow bugs is that use of old code especially ALP & COBOL code are dominated by control flow bugs.

Control and sequence bugs at all levels are caught by testing, especially structural testing, more specifically path testing combined with a bottom line functional test based on a specification.

Logic Bugs:

Bugs in logic, especially those related to misunderstanding how case statements and logic operators behave singly and combinations Also includes evaluation of boolean expressions in deeply nested IF-THEN-ELSE constructs. If the bugs are parts of logical (i.e. boolean) processing not related to control flow, they are characterized as processing bugs.

If the bugs are parts of a logical expression (i.e control-flow statement) which is used to direct the control flow, then they are categorized as control-flow bugs.

Processing Bugs:

Processing bugs include arithmetic bugs, algebraic, mathematical function evaluation, algorithm selection and general processing. Examples of Processing bugs include: Incorrect conversion from one data representation to other, ignoring overflow, improper use of grater-than-or-eual etc Although these bugs are frequent (12%), they tend to be caught in good unit testing.

Initialization Bugs:

Initialization bugs are common. Initialization bugs can be improper and superfluous.

Superfluous bugs are generally less harmful but can affect performance. Typical initialization bugs include: Forgetting to initialize the variables before first use, assuming that they are initialized elsewhere, initializing to the wrong format, representation or type etc

Explicit declaration of all variables, as in Pascal, can reduce some initialization problems.

Data-Flow Bugs and Anomalies:

Most initialization bugs are special case of data flow anamolies. A data flow anomaly occurs where there is a path along which we expect to do something unreasonable with data, such as using an uninitialized variable, attempting to use a variable before it exists, modifying and then not storing or using the result, or initializing twice without an intermediate use.

DATA BUGS:

Data bugs include all bugs that arise from the specification of data objects, their formats, the number of such objects, and their initial values. Data Bugs are atleast as common as bugs in code, but they are foten treated as if they didnot exist at all.

Code migrates data: Software is evolving towards programs in which more and more of the control and processing functions are stored in tables. Because of this, there is an increasing awareness that bugs in code are only half the battle and the data problems should be given equal attention. Dynamic data are transitory. Whatever their purpose their lifetime is relatively short, typically the processing time of one transaction. A storage object may be used to hold dynamic data of different types, with different formats, attributes and residues.

Dynamic data bugs are due to leftover garbage in a shared resource. This can be handled in one of the three ways: (1) Clean up after the use by the user (2) Common Cleanup by the resource manager (3) No Clean up Static Data are fixed in form and content. They appear in the source code or database directly or indirectly, for example a number, a string of characters, or a bit pattern. Compile time processing will solve the bugs caused by static data.

Information, parameter, and control: Static or dynamic data can serve in one of three roles, or in combination of roles: as a parameter, for control, or for information.

Content, Structure and Attributes: Content can be an actual bit pattern, character string, or number put into a data structure. Content is a pure bit pattern and has no meaning unless it is interpreted by a hardware or software processor. All data bugs result in the corruption or misinterpretation of content. **Structure** relates to the size, shape and numbers that describe the data object, that is memory location used to store the content. (e.g A two dimensional array). **Attributes** relates to the specification meaning that is the semantics associated with the contents of a data object. (e.g. an integer, an alphanumeric string, a subroutine). *The severity and subtlelty of bugs increases as we go from content to attributes because the things get less formal in that direction.*

CODING BUGS:

Coding errors of all kinds can create any of the other kind of bugs.

Syntax errors are generally not important in the scheme of things if the source language translator has adequate syntax checking.

If a program has many syntax errors, then we should expect many logic and coding bugs.

The documentation bugs are also considered as coding bugs which may mislead the maintenance programmers.

Various categories of bugs in Interface, Integration, and System Bugs are:

External Interfaces:

The external interfaces are the means used to communicate with the world. These include devices, actuators, sensors, input terminals, printers, and communication lines. The primary design criterion for an interface with outside world should be robustness. All external interfaces, human or machine should employ a protocol. The protocol may be wrong or incorrectly implemented. Other external interface bugs are: invalid timing or sequence assumptions related to external signals. Misunderstanding external input or output formats. Insufficient tolerance to bad input data.

Internal Interfaces: Internal interfaces are in principle not different from external interfaces but they are more controlled. A best example for internal interfaces are communicating routines. The external environment is fixed and the system must adapt to it but the internal environment, which consists of interfaces with other components, can be negotiated. Internal interfaces have the same problem as external interfaces.

Hardware Architecture:

Bugs related to hardware architecture originate mostly from misunderstanding how the hardware works. Examples of hardware architecture bugs: address generation error, i/o device operation / instruction error, waiting too long for a response, incorrect interrupt handling etc.

The remedy for hardware architecture and interface problems is two fold: (1) Good Programming and Testing (2) Centralization of hardware interface software in programs written by hardware interface specialists.

Operating System Bugs:

Program bugs related to the operating system are a combination of hardware architecture and interface bugs mostly caused by a misunderstanding of what it is the operating system does. Use operating system interface specialists, and use explicit interface modules or macros for all operating system calls. This approach may not eliminate the bugs but at least will localize them and make testing easier.

Software Architecture:

Software architecture bugs are the kind that called - interactive.

Routines can pass unit and integration testing without revealing such bugs. Many of them depend on load, and their symptoms emerge only when the system is stressed.

Sample for such bugs: Assumption that there will be no interrupts, Failure to block or un block interrupts, Assumption that memory and registers were initialized or not initialized etc Careful integration of modules and subjecting the final system to a stress test are effective methods for these bugs.

Control and Sequence Bugs (Systems Level):

These bugs include: Ignored timing, Assuming that events occur in a specified sequence, Working on data before all the data have arrived from disc, Waiting for an impossible combination of prerequisites, Missing, wrong, redundant or superfluous process steps. The remedy for these bugs is highly structured sequence control. Specialize, internal, sequence control mechanisms are helpful.

Resource Management Problems:

Memory is subdivided into dynamically allocated resources such as buffer blocks, queue blocks, task control blocks, and overlay buffers. External mass storage units such as discs, are subdivided into memory resource pools. Some resource management and usage bugs: Required resource not obtained, Wrong resource used, Resource is already in use, Resource dead lock etc

Resource Management Remedies: A design remedy that prevents bugs is always preferable to a test method that discovers them. The design remedy in resource management is to keep the resource structure simple: the fewest different kinds of resources, the fewest pools, and no private resource management

Integration Bugs:

Integration bugs are bugs having to do with the integration of, and with the interfaces between, working and tested components. These bugs results from inconsistencies or incompatibilities between components. The communication methods include data structures, call sequences, registers, semaphores, communication links and protocols results in integration bugs.

The integration bugs do not constitute a big bug category(9%) they are expensive category because they are usually caught late in the game and because they force changes in several components and/or data structures.

System Bugs:

System bugs covering all kinds of bugs that cannot be ascribed to a component or to their simple interactions, but result from the totality of interactions between many components such as programs, data, hardware, and the operating systems. There can be no meaningful system testing until there has been thorough component and integration testing.

System bugs are infrequent(1.7%) but very important because they are often found only after the system has been fielded.

TEST AND TEST DESIGN BUGS:

Testing: testers have no immunity to bugs. Tests require complicated scenarios and databases. They require code or the equivalent to execute and consequently they can have bugs.

Test criteria: if the specification is correct, it is correctly interpreted and implemented, and a proper test has been designed; but the criterion by which the software's behavior is judged may be incorrect or impossible. So, a proper test criteria has to be designed. The more complicated the criteria, the likelier they are to have bugs.

Remedies:

The remedies of test design are

Test Debugging: The first remedy for test bugs is testing and debugging the tests. Test debugging, when compared to program debugging, is easier because tests, when properly designed are simpler than programs and do not have to make concessions to efficiency.

Test Quality Assurance: Programmers have the right to ask how quality in independent testing is monitored.

Test Execution Automation: The history of software bug removal and prevention is indistinguishable from the history of programming automation aids. Assemblers, loaders, compilers are developed to reduce the incidence of programming and operation errors. Test execution bugs are virtually eliminated by various test execution automation tools.

Test Design Automation: Just as much of software development has been automated, much test design can be and has been automated. For a given productivity rate, automation reduces the bug count - be it for software or be it for tests.

6. PATH-TESTING BASICS

6.1 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.

6.2 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

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 are 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.

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.

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

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

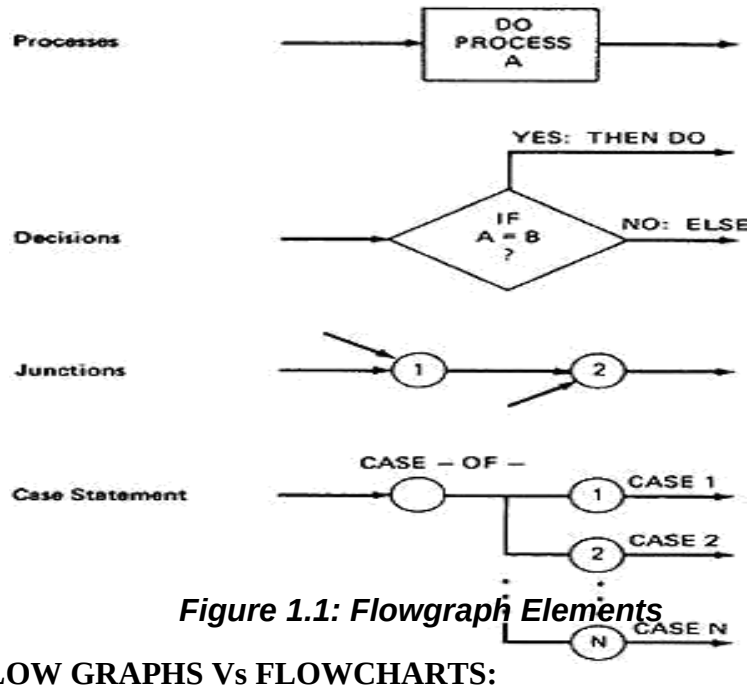


Figure 1.1: Flowgraph Elements

CONTROL FLOW GRAPHS Vs FLOWCHARTS:

A program's flow chart resembles a control flow graph. In flow graphs, we don't show the details of what is in a process block. In flow charts every part of the process block is drawn. The flowchart focuses on process steps, where as the flow graph focuses on control flow of the program. The act of drawing a control flow graph is a useful tool that can help us clarify the control flow and data flow issues.

NOTATIONAL EVOULTION:

The control flow graph is simplified representation of the program's structure. The notation changes made in creation of control flow graphs: The process boxes weren't really needed. There is an implied process on every line joining junctions and decisions. We don't need to know the specifics of the decisions, just the fact that there is a branch. The specific target label names aren't important-just the fact that they exist. So we can replace them by simple numbers.

To understand this, we will go through an example (Figure 1.2) written in a FORTRAN like programming language called **Programming Design Language (PDL)**. The program's corresponding flowchart (Figure 1.3) and flowgraph (Figure 1.4) were also provided below for better understanding.

The first step in translating the program to a flowchart is shown in Figure 1.3, where we have the typical one-for-one classical flowchart. Note that complexity has increased, clarity has decreased, and that we had to add auxiliary labels (LOOP, XX, and YY), which have no actual program counterpart. In Figure 1.4 we merged the process steps and replaced them with the single process box. We now have a control flowgraph. But this representation is still too busy. We simplify the notation further to achieve Figure 1.5, where for the first time we can really see what the control flow looks like.

CODE* (PDL)

```
      INPUT X, Y                      V(U−1):=V(U+1) + U(V−1)
      Z := X + Y                ELL:V(U+U(V)) := U + V
      V := X − Y                      IF U = V GOTO JOE
      IF Z >=Ø GOTO SAM               IF U > V THEN U := Z
JOE:  Z := Z − 1                      Z := U
SAM:  Z := Z + V                      END
      FOR U = Ø TO Z
      V(U),U(V) := (Z + V)*U
      IF V(U)= Ø GOTO JOE
      Z := Z − 1
      IF Z = Ø GOTO ELL
      U := U + 1
      NEXT U
```

* A contrived horror

Figure 1.2: Program Example (PDL)

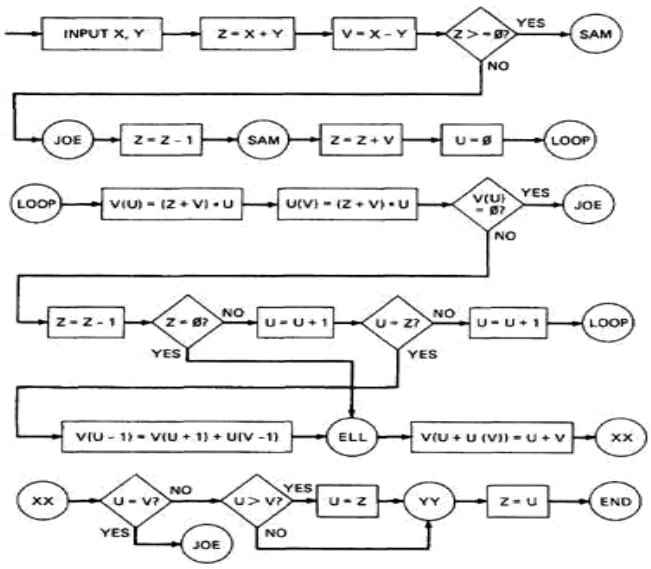


Figure 1.3: One-to-one flowchart for example program in Figure1.2

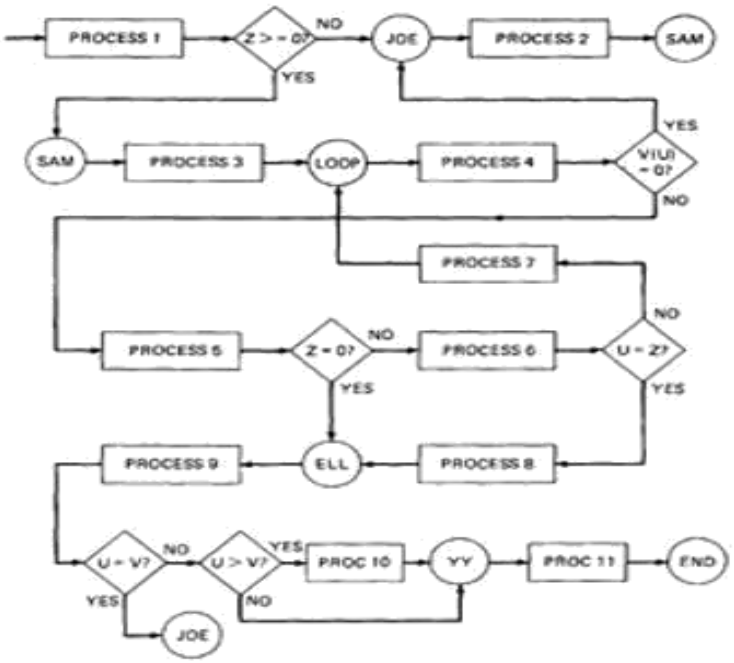


Figure 1.4: Control Flowgraph for example in Figure 1.2

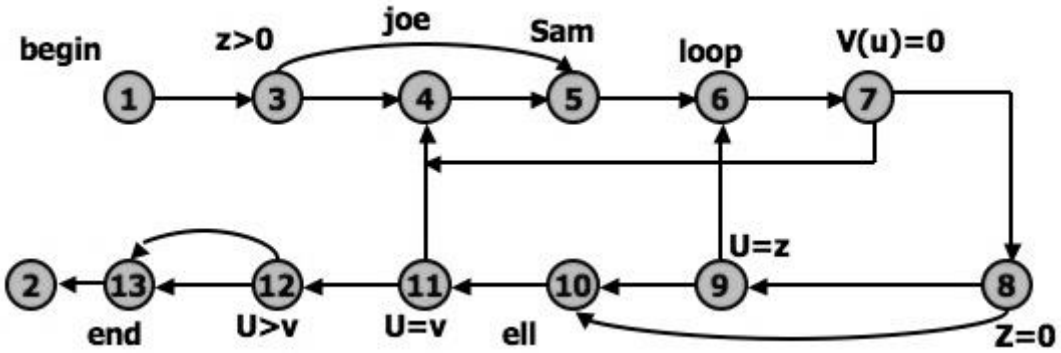


Figure 1.5: Simplified Flowgraph Notation

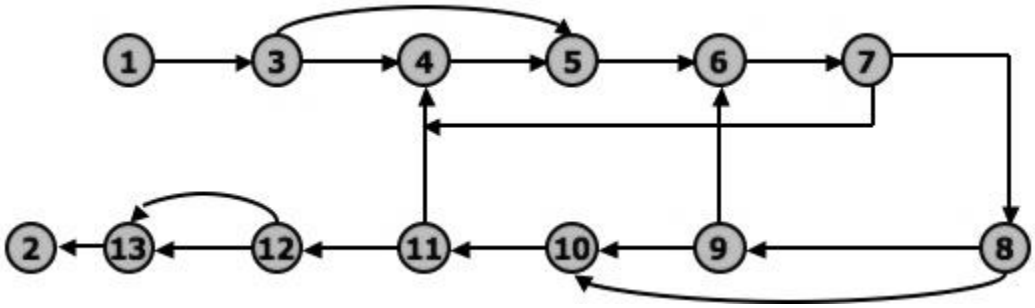


Figure 1.6: Even Simplified Flowgraph Notation

The final transformation is shown in Figure 1.6, where we've dropped the node numbers to achieve an even simpler representation. The way to work with control flowgraphs is to use the simplest possible representation - that is, no more information than you need to correlate back to the source program or PDL

LINKED LIST REPRESENTATION:

Although graphical representations of flow graphs are revealing, the details of the control flow inside a program they are often inconvenient.
In linked list representation, each node has a name and there is an entry on the list for each link in the flow graph. only the information pertinent to the control flow is shown.

```
 1 (BEGIN)    : 3
 2 (END)      :              Exit, no outlink
 3 (Z>Ø?)     : 4 (FALSE)
              : 5 (TRUE)
 4 (JOE)      : 5
 5 (SAM)      : 6
 6 (LOOP)     : 7
 7 (V(U)=Ø?)  : 4 (TRUE)
              : 8 (FALSE)
 8 (Z=Ø?)     : 9 (FALSE)
              :10 (TRUE)
 9 (U=Z?)     : 6 (FALSE) = LOOP
              :10 (TRUE) = ELL
10 (ELL)      :11
11 (U=V?)     : 4 (TRUE) = JOE
              :12 (FALSE)
12 (U>V?)     :13 (TRUE)
              :13 (FALSE)
13            : 2 (END)
```

Figure 1.7: Linked List Control Flowgraph Notation

FLOWGRAPH - PROGRAM CORRESPONDENCE:

A flow graph is a pictorial representation of a program and not the program itself, just as a topographic map. You cant always associate the parts of a program in a unique way with flowgraph parts because many program structures, such as if-then-else constructs, consists of a combination of decisions, junctions, and processes.

The translation from a flowgraph element to a statement and vice versa is not always unique. (See Figure 1.8)

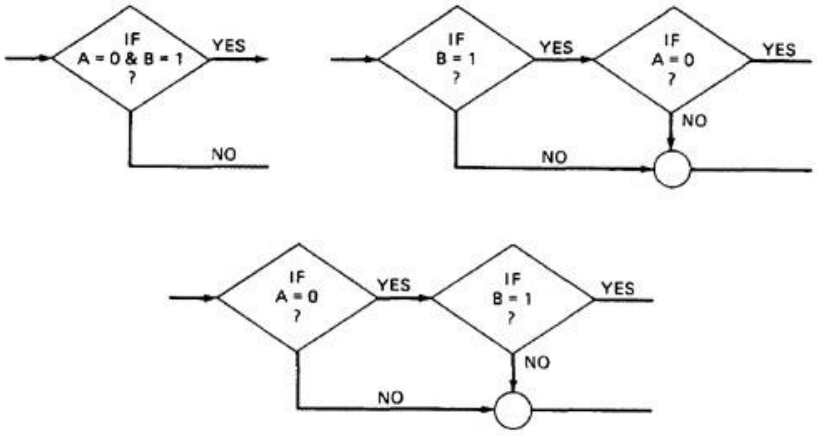


Figure 1.8: Alternative Flowgraphs for same logic (Statement "IF (A=0) AND (B=1) THEN . . .").

An improper translation from flowgraph to code during coding can lead to bugs, and improper translation during the test design lead to missing test cases and causes undiscovered bugs

FLOWGRAPH AND FLOWCHART GENERATION:

Flowcharts can be

1. Handwritten by the programmer.
2. Automatically produced by a flowcharting program based on a mechanical analysis of the source code.
3. Semi automatically produced by a flow charting program based in part on structural analysis of the source code and in part on directions given by the programmer.

There are relatively few control flow graph generators.

6.3 PATH TESTING - PATHS, NODES AND LINKS:

Path: A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same junction, decision, or exit. A path may go through several junctions, processes, or decisions, one or more times.

- Paths consists of segments.
- The segment is a link - a single process that lies between two nodes.
- A path segment is succession of consecutive links that belongs to some path.
- The length of path measured by the number of links in it and not by the number of the instructions or statements executed along that path.
- The name of a path is the name of the nodes along the path.

Multi Entry/Multi-Exit Routines

If a routine can have several different kinds of outcomes then an exit parameter should be used.

Instead of using direct linkages between multiple exits and entrances we handle the control flow by examining the values of exit parameter that can serve as entry parameter to next routine

The trouble with the multi entry or multi exit routines is that it can be very difficult to determine what the inter process control flow is and consequently it is easy to miss some important test cases

FUNDAMENTAL PATH SELECTION CRITERIA:

There are many paths between the entry and exit of a typical routine.

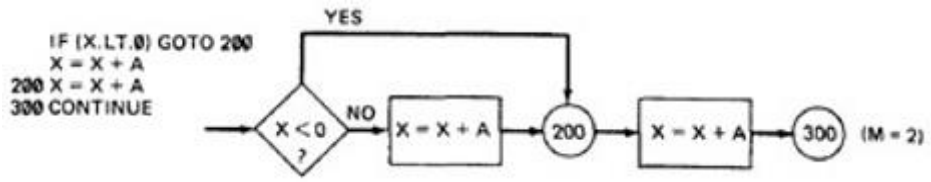
Every decision doubles the number of potential paths. And every loop multiplies the number of potential paths by the number of different iteration values possible for the loop.

Defining complete testing:

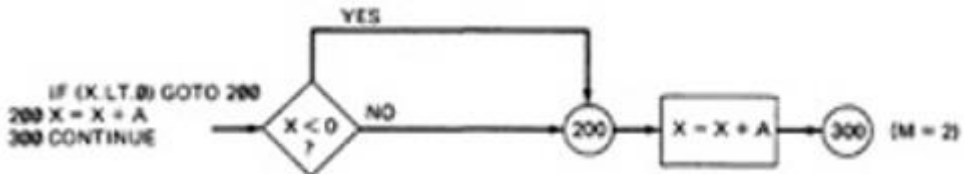
1. Exercise every path from entry to exit
2. Exercise every statement or instruction at least once
3. Exercise every branch and case statement, in each direction at least once

If prescription 1 is followed then 2 and 3 are automatically followed. But it is impractical for most routines. It can be done for the routines that have no loops, in which it is equivalent to 2 and 3 prescriptions.

EXAMPLE: **Here is the correct version.**

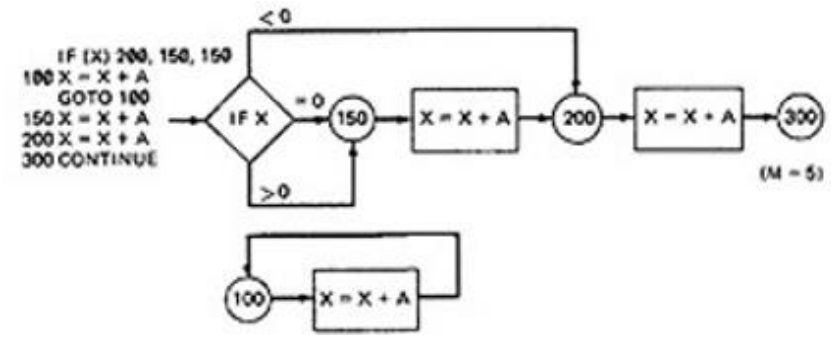


For X negative, the output is X + A, while for X greater than or equal to zero, the output is X + 2A. Following prescription 2 and executing every statement, but not every branch, would not reveal the bug in the following incorrect version:



A negative value produces the correct answer. Every statement can be executed, but if the test cases do not force each branch to be taken, the bug can remain

hidden. The next example uses a test based on executing each branch but does not force the execution of all statements:



The hidden loop around label 100 is not revealed by tests based on prescription 3 alone because no test forces the execution of statement 100 and the following GOTO statement. Furthermore label 100 is not flagged by the compiler as an unreferenced label and the subsequent GOTO does not refer to an undefined label.

A **Static Analysis** (that is, an analysis based on examining the source code or structure) cannot determine whether a piece of code is or is not reachable. There could be subroutine calls with parameters that are subroutine labels, or in the above example there could be a GOTO that targeted label 100 but could never achieve a value that would send the program to that label.

A **Dynamic Analysis** (that is, an analysis based on the code's behavior while running - which is to say, to all intents and purposes, testing) can determine whether code is reachable or not and therefore distinguish between the ideal structure we think we have and the actual, buggy structure.

PATH TESTING CRITERIA:

Any testing strategy based on paths must at least both exercise every instruction and take branches in all directions.

A set of tests that does this is not complete in an absolute sense, but it is complete in the sense that anything less must leave something untested.

So we have explored three different testing criteria or strategies out of a potentially infinite family of strategies.

1. **Path Testing (P_{inf})**:

Execute all possible control flow paths through the program: typically, this is restricted to all possible entry/exit paths through the program.

If we achieve this prescription, we are said to have achieved 100% path coverage. This is the strongest criterion in the path testing strategy family: it is generally impossible to achieve.

2. **Statement Testing (P_1)**:

- Execute all statements in the program at least once under some test. If we do enough tests to achieve this, we are said to have achieved 100% statement coverage.
- An alternate equivalent characterization is to say that we have achieved 100% node coverage. We denote this by C1.

- This is the weakest criterion in the family: testing less than this for new software is unconscionable (unprincipled or can not be accepted) and should be criminalized.

3. **Branch Testing (P_2)**:

Execute enough tests to assure that every branch alternative has been exercised at least once under some test.

If we do enough tests to achieve this prescription, then we have achieved 100% branch coverage.

An alternative characterization is to say that we have achieved 100% link coverage

For structured software, branch testing and therefore branch coverage strictly includes statement coverage. We denote branch coverage by C2.

Commonsense and Strategies:

Branch and statement coverage are accepted today as the minimum mandatory testing requirement.

In the view of common sense and experience since:

(1) Not testing a piece of a code leaves a residue of bugs in the program in proportion to the size of the untested code and the probability of bugs.

(2) The high probability paths are always thoroughly tested if only to demonstrate that the system works properly.

Which paths to be tested?

You must pick enough paths to achieve C1+C2. The question of what is the fewest number of such paths is interesting to the designer of test tools that help automate the path testing, but it is not crucial to the pragmatic (practical) design of tests. It is better to make many simple paths than a few complicated paths.

Path Selection Example

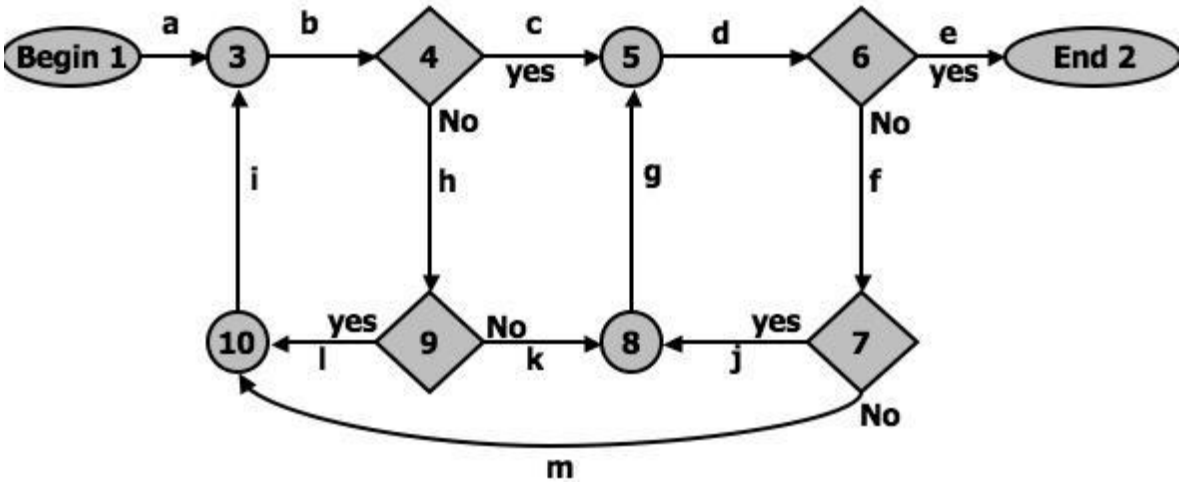


Figure 2.1 :An example flow graph to explain path selection

Practical Suggestions in Path Testing:

1. Draw the control flow graph on a single sheet of paper.

2. Make several copies - as many as you will need for coverage (C1+C2) and several more.

3. Use a yellow highlighting marker to trace paths. Copy the paths onto master sheets.

4. Continue tracing paths until all lines on the master sheet are covered, indicating that you appear to have achieved C1+C2.

5. As you trace the paths, create a table that shows the paths, the coverage status of each process, and each decision.

6. The above paths lead to the following table considering Figure 2.1.

PATHS	DECISIONS				PROCESS–LINK												
	4	6	7	9	a	b	c	d	e	f	g	h	i	j	k	l	m
abcde	YES	YES			✓	✓	✓	✓	✓								
abhkgde	NO	YES		NO	✓	✓		✓	✓		✓	✓			✓		
abhlibcde	NO,YES	YES		YES	✓	✓	✓	✓	✓			✓	✓			✓	
abcdfjgde	YES	NO,YES	YES		✓	✓	✓	✓	✓	✓	✓			✓			
abcdfmibcde	YES	NO,YES	NO		✓	✓	✓	✓	✓	✓			✓				✓

7. After you have traced a covering path set on the master sheet and filled in the table for every path, check the following:

- ✓ Does every decision have a YES and a NO in its column? (C2)
- ✓ Has every case of all case statements been marked? (C2)
- ✓ Is every three - way branch (less, equal, greater) covered? (C2)
- ✓ Is every link (process) covered at least once? (C1)

8**. Revised Path Selection Rules:**

- ✓ Pick the simplest, functionally sensible entry/exit path.
- ✓ Pick additional paths as small variation from previous paths. Pick paths that do not have loops rather than paths that do. Favor short paths that make sense over paths that don't.
- ✓ Pick additional paths that have no obvious functional meaning only if it's necessary to provide coverage.
- ✓ Be comfortable with your chosen paths. Play your hunches (guesses) and give your intuition free reign as long as you achieve C1+C2.

6.4 LOOPS:

Cases for a single loop: A Single loop can be covered with two cases: Looping and Not looping. But, experience shows that many loop-related bugs are not discovered by C1+C2. Bugs hide themselves in corners and congregate at boundaries - in the cases of loops, at or around the minimum or maximum number of times the loop can be iterated. The minimum number of iterations is often zero, but it need not be.

CASE 1: Single loop, Zero minimum, N maximum, No excluded values

1. Try bypassing the loop (zero iterations). If you can't, you either have a bug, or zero is not the minimum and you have the wrong case.

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.

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. What prevents the loop-control variable from having this value? What will happen with this value if it is forced?

CASE 2: Single loop, Non-zero minimum, No excluded values

1. Try one less than the expected minimum. What happens if the loop control variable'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 case 1 and 2 above.
- Example, the total range of the loop control variable was 1 to 20, but that values 7, 8,9,10 were excluded. The two sets of tests are 1-6 and 11-20.
- The test cases to attempt would be 0,1,2,4,6,7 for the first range and 10,11,15,19,20,21 for the second range.

Kinds of Loops: There are only three kinds of loops with respect to path testing:

- **Nested Loops:**

The number of tests to be performed on nested loops will be the exponent of the tests performed on single loops.As we cannot always afford to test all combinations of nested loops' iterations values. Here's a tactic used to discard some of these values:

1. Start at the inner most loop. Set all the outer loops to their 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. Expand the tests as required for out of range and excluded values.

3. If you've done the outmost loop, GOTO step 5, else move out one loop and set it up as in step 2 with all other loops set to typical values.

4. Continue outward in this manner until all loops have been covered.

5. Do all the cases for all loops in the nest simultaneously.

- **Concatenated Loops:**

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

- **Horrible Loops:**

A horrible loop is a combination of nested loops, the use of code that jumps into and out of loops, intersecting loops, hidden loops, and cross connected loops. Makes iteration value selection for test cases an awesome and ugly task, which is another reason such structures should be avoided.

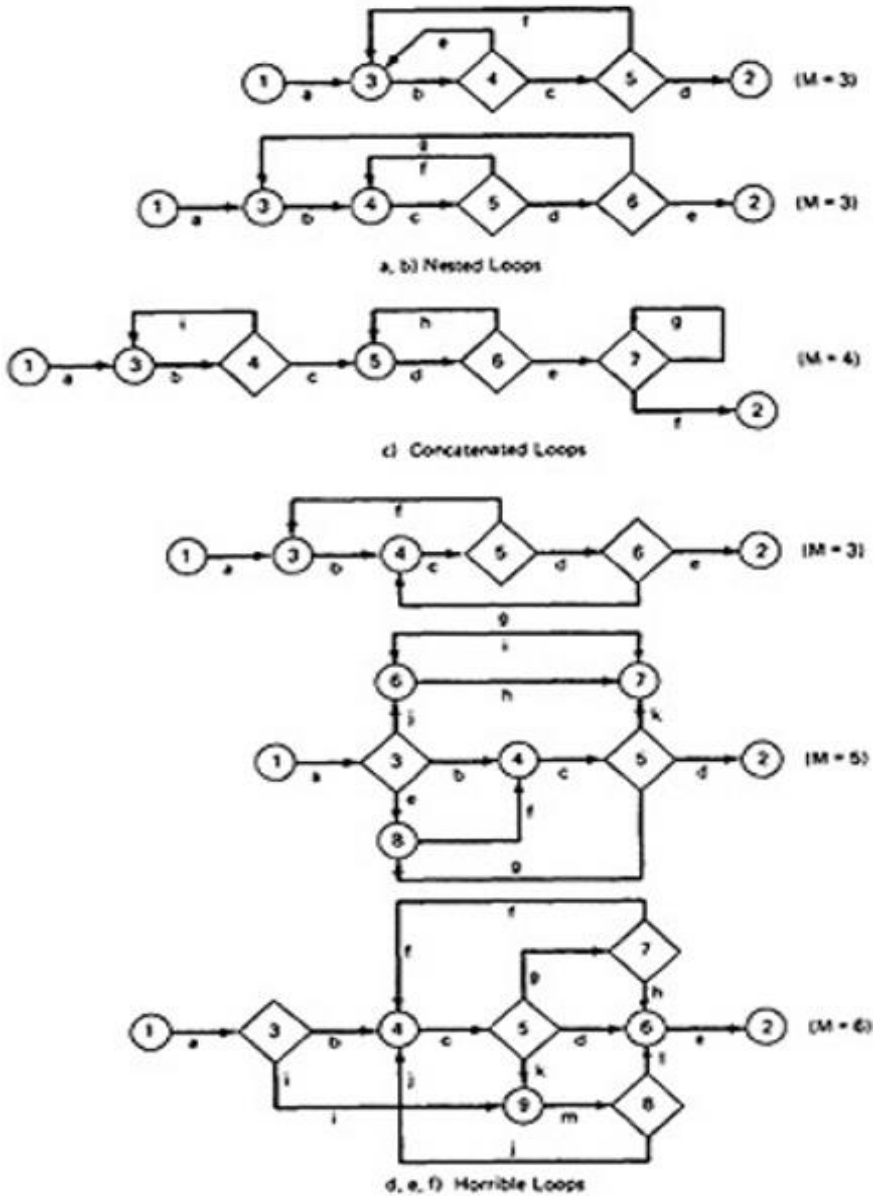


Figure 2.10: Example of Loop types

Loop Testing Time:

Any kind of loop can lead to long testing time, especially if all the extreme value cases are to attempted (Max-1, Max, Max+1).

- This situation is obviously worse for nested and dependent concatenated loops.
- Consider nested loops in which testing the combination of extreme values lead to long test times. Several options to deal with:
- Prove that the combined extreme cases are hypothetically possible, they are not possible in the real world
- Put in limits or checks that prevent the combined extreme cases. Then you have to test the software that implements such safety measures

6.5 More on Testing Multi-Entry/Multi-Exit Routines

Single Entry and single exit routines are preferable because they are called well-formed routines and tests could generate test cases

Multi entry and Multi exit routines are not preferable because they are called ill-formed routines and a good formal basis does not exist and tools may fail to generate important test cases

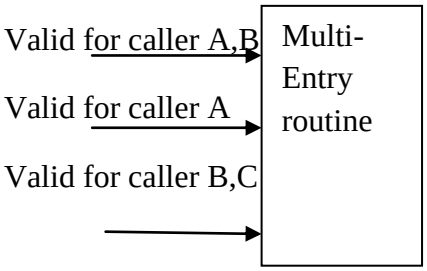


Fig: Example of a Multi entry routine

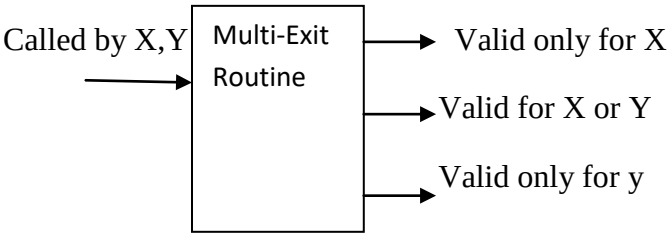


Fig: Example of a Multi exit routine

Multi-Entry/Multi-Exit routines have

1. Large number of inter-process interfaces which creates integration problem
2. More number of test cases and also a formal treatment is more difficult

The only solution is to convert multi entry and multi exit routines in to single entry and single exit routines.

Multi-Entry routine to Single-Entry routine

Use an entry parameter and a case statement at the entry to convert Multi-Entry routine into Single-Entry routine.

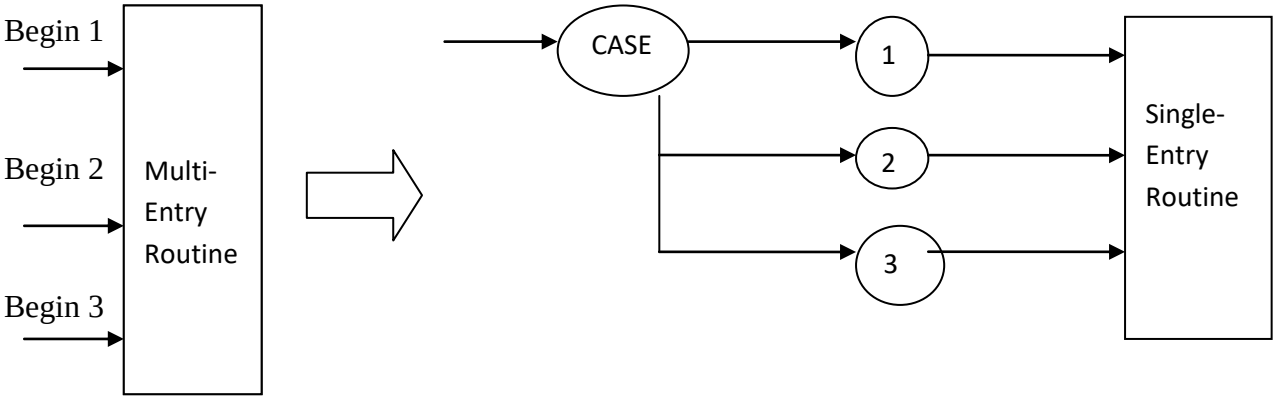


Figure: A Multi-Entry routine is converted into an equivalent single-entry routine

Multi-Exit routine to Single-Exit routine

Merge all exits to single exit point after setting value to each exit parameter

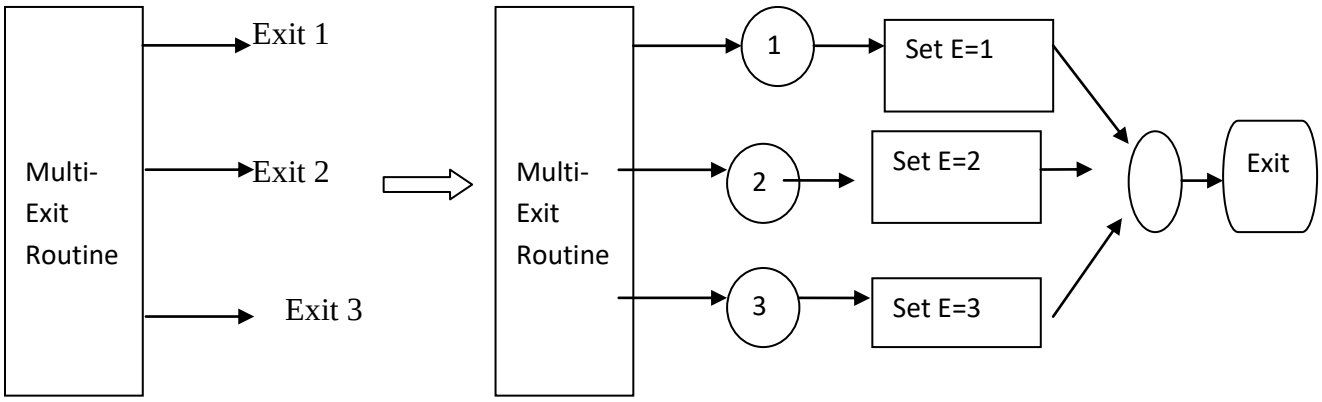


Figure: A Multi-Exit routine is converted into an equivalent single-exit routine

Strategy Summary:

1. Get rid of them

2. Control those you cannot get rid of

3. Convert to single entry/exit routines

4. Do unit testing by treating each entry/exit combination as if it were a completely different routine

5. Recognize that integration testing is heavier

6. Understand the strategies and assumptions in the automatic test generators and confirm that they do or do not work for multi-entry or multi-exit routines

6.6 Effectiveness of Path testing

6.6.1 Effectiveness and limitations

Approximately 65% of all bugs can be caught in unit testing which is dominated by Path-testing methods(Statement and Branch testing)

When Path testing is combined with other methods such as limit checks on loops the percentage of bugs caught rises to 50% to 60% in unit testing. Path testing is more effective for unstructured than for structured software

The statistics indicated that path testing as a sole technique is limited. Here are some of the reasons:

1. Path testing may not cover if you have bugs

2. Path testing may not reveal totally wrong or missing functions

3. Interface errors may not be caught by unit-level path testing

4. Database and Data flow errors may not be caught

5. Initialization errors are not caught by path testing

6. Specification errors can not be caught by path testing

6.6.2 A lot of work:

Creating the flow graph ,selecting the set of covering paths, finding input data values to force those paths, setting up the loop cases and combinations it's a lot of work

Statistics shows that we are spending half of our time in testing and debugging

Furthermore the act of careful, complete, systematic, test design will catch as many bugs as the act of testing

The fact that bugs caught during test design cost less to fix than bugs caught during testing

6.6.3 More on how to do it:

The only tools needed for path testing are

1. Source code testing

2. Yellow marking pen

3. Copying Machine

- Initially create a control flow graph and use it as basis for test design.
- As we gain experience with practice select paths directly on the source code without bothering to draw the control flow graph
- Designing test with code is almost the same way as you would with a pictorial control flow graph
- Make several copies of the source
- Select your path, marking the statements on the path with in the marking pen
- Translate your markings to a master sheet
- When it's all yellow you have a covering path set
- Check that you have achieved c1+c2

6.6.4 Variations:

Branch and statement coverage as basic teaching criteria are well established as effective, reasonable and easy to implement

How about the more complicated test criteria in the path testing family

There are two main classes of variations

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

7. PREDICATES, PATH PREDICATES AND ACHIEVABLE PATHS

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

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

PREDICATE INTERPRETATION:

- The simplest predicate depends only on input variables.

- For example if x1,x2 are inputs, the predicate might be x1+x2>=7, given the values of x1 and x2 the direction taken through the decision is based on the predicate is determined at input time and does not depend on processing.

- Another example, assume a predicate x1+y>=0 that along a path prior to reaching this predicate we had the assignment statement y=x2+7. although our predicate depends on processing, we can substitute the symbolic expression for y to obtain an equivalent predicate x1+x2+7>=0.

- The act of symbolic substitution of operations along the path in order to express the predicate solely in terms of the input vector is called **predicate interpretation.**

- Sometimes 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 multiway branch. It yields for the three cases respectively, if Y+7>0, Y-7>0, Y>0.

- The path predicates are the specific form of the predicates of the decisions along 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:

X1+3X2+17>=0
X3=17
X4-X1>=14X2

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

- Sometimes a predicate can have an OR in it.

- Example:

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

Boolean algebra notation to denote the boolean expression:

ABCD+EBCD=(A+E)BCD

PREDICATE COVERAGE:

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

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

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

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

TESTING BLINDNESS:

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

- There are three types of Testing Blindness:

1. **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 ..... if Y > 0 then ...	X = 7 ..... if X+Y > 0 then ...

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

2. **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.

3. **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

8. PATH SENSITIZING:

REVIEW: ACHIEVABLE AND UNACHIEVABLE PATHS:

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

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

5. Multiply out the expression to achieve a sum of products form:
   ADFGHIJKL+AEFGHIJKL+BCDFGHIJKL+BCEFGHIJKL

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

7. 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 cant find a solution to any of the sets of inequalities, the path is un achievable.

- ✓ The act of finding a set of solutions to the path predicate expression is called **PATH SENSITIZATION**.

HEURISTIC PROCEDURES FOR SENSITIZING PATHS:

1. This is a workable approach, instead of selecting the paths without considering how to sensitize, attempt to choose a covering path set that is easy to sensitize and pick hard to sensitize paths only as you must to achieve coverage.
2. Identify all variables that affect the decision.
3. Classify the predicates as dependent or independent.
4. Start the path selection with un correlated, independent predicates.
5. If coverage has not been achieved using independent uncorrelated predicates, extend the path set using correlated predicates.
6. If coverage has not been achieved extend the cases to those that involve dependent predicates.
7. Last, use correlated, dependent predicates

Examples for Path Sensitization

1. Simple Independent Uncorrelated Predicates

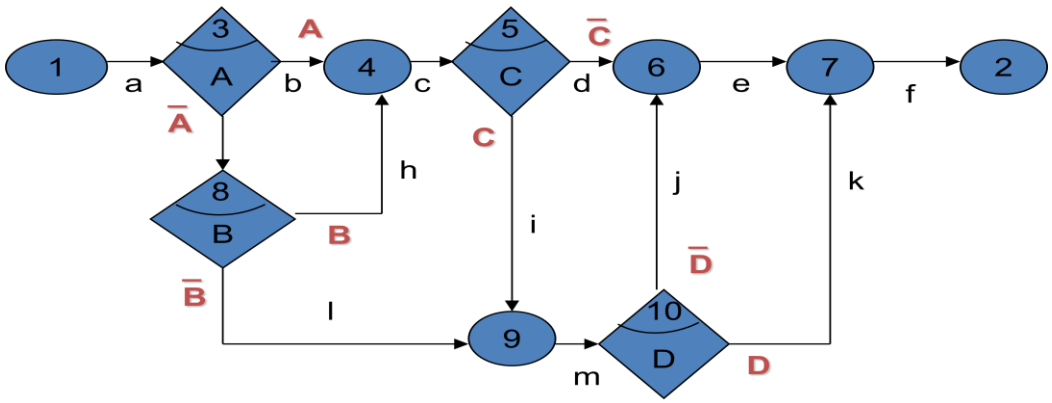


Figure: Predicate Notation

Path	**Predicate values**
abcdef	A $\bar{C}$
abcimkf	A CD
aghcdef	$\bar{A}$ B $\bar{C}$
aglmkf	$\bar{A}$ $\bar{B}$ $\bar{D}$

2. Independent Correlated Predicates

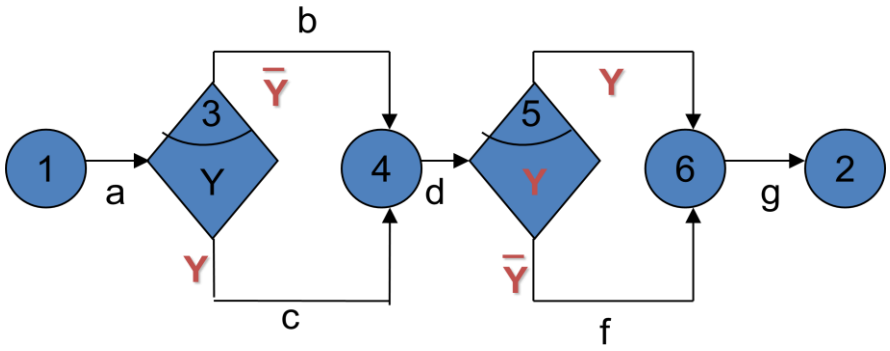


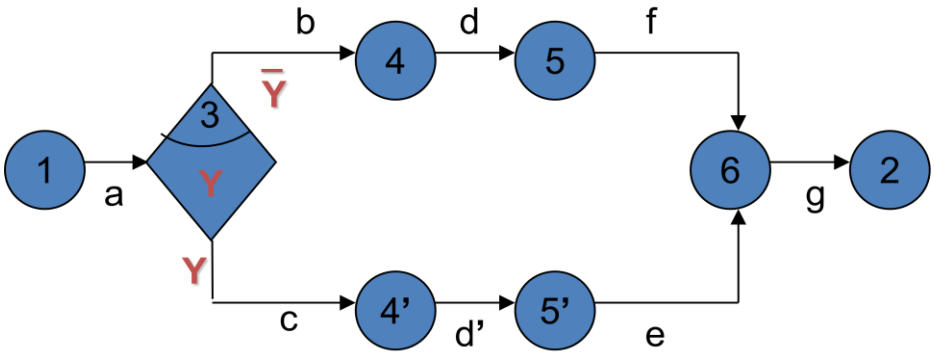
Figure: Correlated Decisions

Correlated paths => **some** paths are unachievable

i.e., redundant paths.
i.e., n decisions but # paths are < 2^n
Due to practice of saving code which makes the code very difficult to maintain.

Eliminate the correlated decisions.

Reproduce common code.



Correlated decision removed & CFG simplified

If a chosen sensible path is not achievable,
- there's a bug.
- design can be simplified.
- get better understanding of correlated decisions

3. Dependent Predicates

Usually most of the processing does not affect the control flow. Use computer simulation for sensitization in a simplified way.

Dependent predicates contain iterative loop statements usually.

For Loop statements:

Determine the value of loop control variable for a certain number of iterations, and then

work backward to determine the value of input variables (input vector)

4.The General Case

No simple procedure to solve for values of input vector for a selected path.

1. Select cases to provide coverage on the basis of **functionally sensible paths**.
   Well structured routines allow easy sensitization. Intractable paths may have a bug.
2. Tackle the path with the fewest decisions first. Select paths with least # of loops
3. Start at the end of the path and list the predicates while tracing the path in **reverse**. Each predicate imposes restrictions on the subsequent (in reverse order) predicate.
4. Continue tracing along the path. Pick the broadest range of values for variables affected and consistent with values that were so far determined.
5. Continue until the entrance & therefore have established a set of input conditions for the path.
6. If the solution is not found, path is not achievable, *it could be a bug.*

Alternately:

1. In the **forward** direction, list the decisions to be traversed. For each decision list the broadest range of input values.
2. Pick a path & adjust all input values. These restricted values are used for next decision.
3. Continue. Some decisions may be dependent on and/or correlated with earlier ones.
4. The path is unachievable if the input values become contradictory, or, impossible. If the path is achieved, try a new path for additional coverage.

9. 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

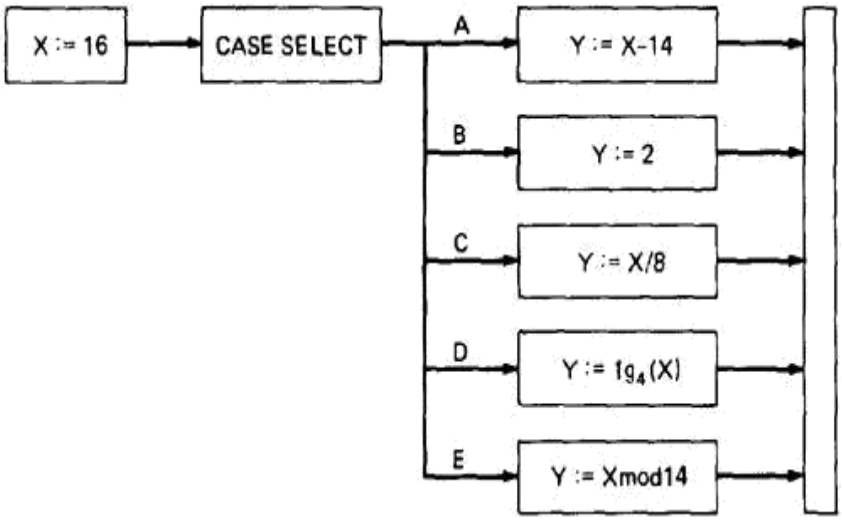


Figure 1.11: Coincidental Correctness

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

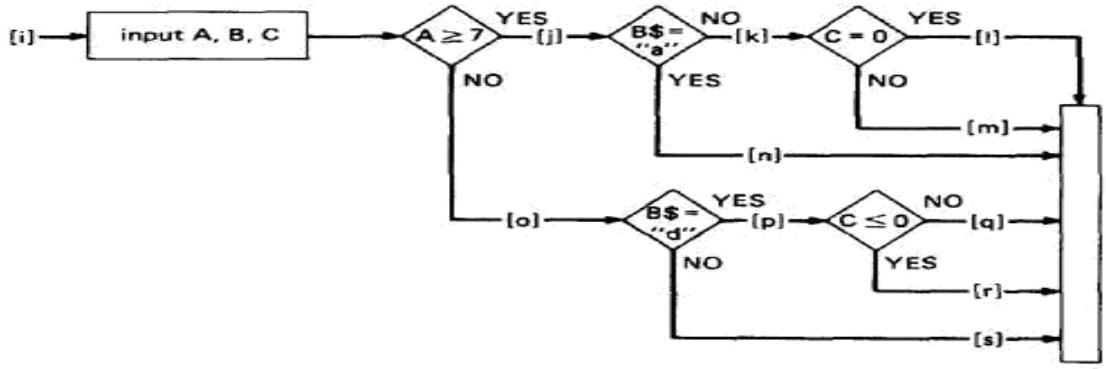


Figure 1.12: Single Link Marker Instrumentation

Why Single Link Markers aren't enough: Unfortunately, a single link marker may not do the trick because links can be chewed by open bugs.

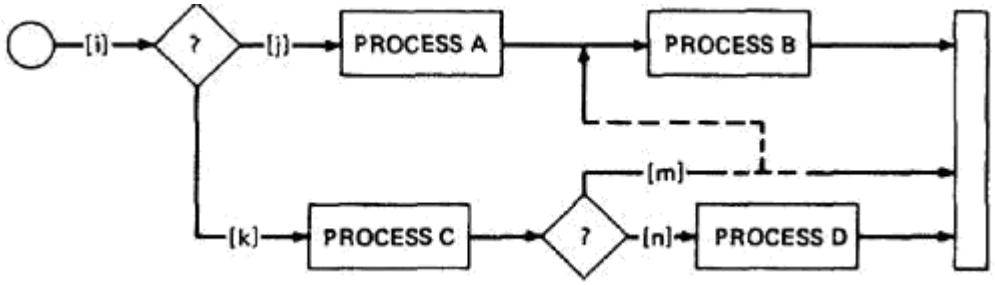


Figure 2.13: Why Single Link Markers aren't enough.

We intended to traverse the ikm path, but because of a rampaging GOTO in the middle of the m link, we go to process B. If coincidental correctness is against us, the outcomes will be the same and we won't know about the bug.

Two Link 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 on at the end.
The two link markers now specify the path name and confirm both the beginning and end of the link

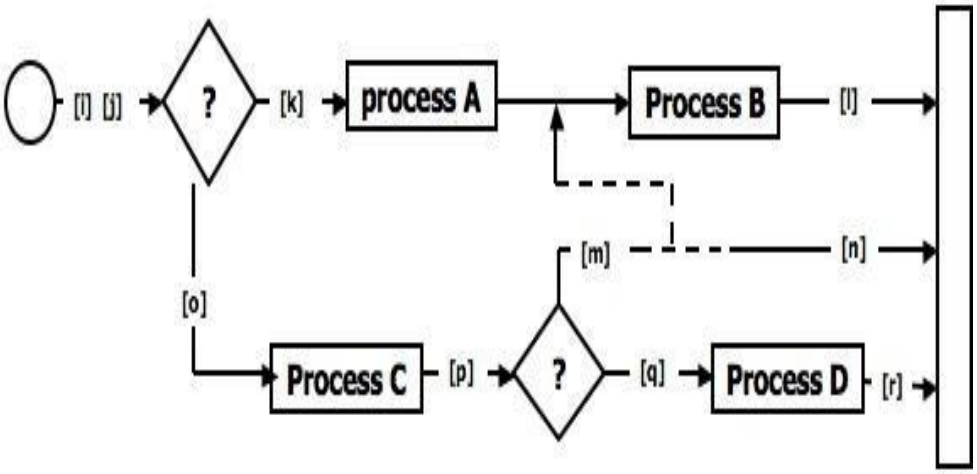


Figure 2.14: Double Link Marker Instrumentation.

Link Counter:

- ✓ Less disruptive and less informative.
- ✓ Increment a link counter each time a link is traversed. **Path length could confirm the intended path**
- ✓ For avoiding the same problem as with markers, use double link counters.
  Expect an even count = double the length.
- ✓ Now, put a link counter on every link. *(earlier it was only between decisions)*
  If there are no loops, the link counts are = 1.
- ✓ Sum the link counts over a series of tests, say, a covering set. Confirm the total link counts with the expected.
- ✓ Using double link counters avoids the same & earlier mentioned problem.

Check list for the procedure:

- ✓ Do begin-link counter values equal the end-link counter values?
- ✓ Does the input-link count of every decision equal to the sum of the link counts of the output links from that decision?
- ✓ Do the sum of the input-link counts for a junction equal the output-link count for that junction?
- ✓ Do the total counts match the values you predicted when you designed the covering test set?

This procedure and the checklist could solve the problem of Instrumentation.

Limitations

- Instrumentation probe (marker, counter) **may disturb the timing relations** & hide racing condition bugs.
- Instrumentation probe (marker, counter) **may not detect location dependent bugs**.

- If the presence or absence of probes modifies things (for example in the data base) in a faulty way, then the **probes hide the bug** in the program.