Software Testing Methodologies-Syllabus

UNIT-I

Introduction:-Purpose of testing, Dichotomies, model for testing, consequences of bugs, taxonomy of bugs, Flow graphs and Path testing:- Basics concepts of path testing, predicates, path predicates and achievable paths, path sensitizing, path instrumentation, application of path testing.

UNIT-II

Transaction Flow Testing:-transaction flows, transaction flow testing techniques. Dataflow testing:- Basics of dataflow testing, strategies in dataflow testing, application of dataflow testing. Domain Testing:-domains and paths, Nice & ugly domains, domain testing, domains and interfaces testing, domain and interface testing, domains and testability.

UNIT-III

Paths, Path products and Regular expressions:- path products &path expression, reduction procedure, applications, regular expressions & flow anomaly detection. Logic Based Testing:-overview, decision tables, path expressions, kv charts, specifications.

UNIT-IV:

State, State Graphs and Transition testing:- state graphs, good & bad state graphs, state testing, Testability tips.

UNIT-V:

Graph Matrices and Application:-Motivational overview, matrix of graph, relations, power of a matrix, node reduction algorithm, building tools

TEXT BOOKS

Software Testing techniques – Boris Beizer, Dreamtech, second edition.
Software Testing Tools – Dr.K.V.K.K.Prasad, Dreamtech.

REFERENCES BOOKS:

The craft of software testing – Brian Marick, Pearson Education.
Software Testing Techniques – SPD(Oreille)
Software Testing in the Real World – Edward Kit, Pearson.
Effective methods of Software Testing, Perry, John Wiley.
Art of Software Testing – Meyers, John Wiley.

UNIT- I

UNIT-I

Introduction:-Purpose of testing, Dichotomies, model for testing, consequences of bugs, taxonomy of bugs,Flow graphs and Path testing:- Basics concepts of path testing, predicates, path predicates and achievable paths, path sensitizing, path instrumentation, application of path testing.

What is testing?

Testing is the process of exercising or evaluating a system or system components by manual or automated means to verify that it satisfies specified requirements.

The Purpose of Testing

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

MYTH: Good programmers write code without bugs. (It's wrong!!!)
History says that even well written programs still have 1-3 bugs per hundred statements.

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 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 tradeoff 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. Whereas the manufacturing cost of 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.

Testing and Test Design are parts of quality assurance should also focus on bug prevention. A prevented bug is better than a detected and corrected bug.

Phases in a tester's mental life:

Phases in a tester's mental life can be categorized into the following 5 phases:

Phase 0: (Until 1956: Debugging Oriented) 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: (1957-1978: Demonstration Oriented) 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 will find a bug.

Phase 2: (1979-1982: Destruction Oriented) the purpose of testing is to show that software doesn't work. This also failed because the software will never get released as you will find one bug or the other. Also, a bug corrected may also lead to another bug.

Phase 3: (1983-1987: Evaluation Oriented) the purpose of testing is not to prove anything but to reduce the perceived risk of not working to an acceptable value (Statistical Quality Control). Notion is that testing does improve the product to the extent that testing catches bugs and to the extent that those bugs are fixed. 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: (1988-2000: Prevention Oriented) Testability is the factor considered here. One reason is to reduce the labor 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.

Testing isn't everything:

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

Inspection Methods: Methods like walkthroughs, desk checking, formal inspections and code reading appear to be as effective as testing but the bugs caught don't 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.

Dichotomies:

Testing Versus Debugging:
Many people consider both as same. Purpose of testing is to show that a program has bugs. The purpose of testing is to find the error or misconception that led to the program's failure and to design and implement the program changes that correct the error.
Debugging usually follows testing, but they differ as to goals, methods and most important psychology.

The below tab le shows few important differences between testing and debugging.

Testing	**Debugging**
Testing starts with known conditions, uses predefined procedures and has predictable outcomes.	Debugging starts from possibly unknown initial conditions and the end cannot be predicted except statistically.
Testing can and should be planned, designed and scheduled.	Procedure and duration of debugging cannot be so constrained.
Testing is a demonstration of error or apparent correctness.	Debugging is a deductive process.
Testing proves a programmer's failure.	Debugging is the programmer's vindication (Justification).
Testing, as executes, should strive to be predictable, dull, constrained, rigid and inhuman.	Debugging demands intuitive leaps, experimentation and freedom.
Much testing can be done without design knowledge.	Debugging is impossible without detailed design knowledge.
Testing can often be done by an outsider.	Debugging must be done by an insider.
Much of test execution and design can be automated.	Automated debugging is still a dream.

Function versus Structure:

Tests can be designed from a functional or a structural point of view.

In **Functional testing**, the program or system is treated as a black box. It is subjected to inputs, and its outputs are verified for conformance to specified behavior. Functional testing takes the user point of view- bother about functionality and features and not the program's implementation.

In **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.

Both Structural and functional tests are useful, both have limitations, and both target different kinds of bugs. Functional tests can detect all bugs but would take infinite time to do so. Structural tests are inherently finite but cannot detect all errors even if completely executed.

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.

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

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.

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?

MODEL FOR TESTING:

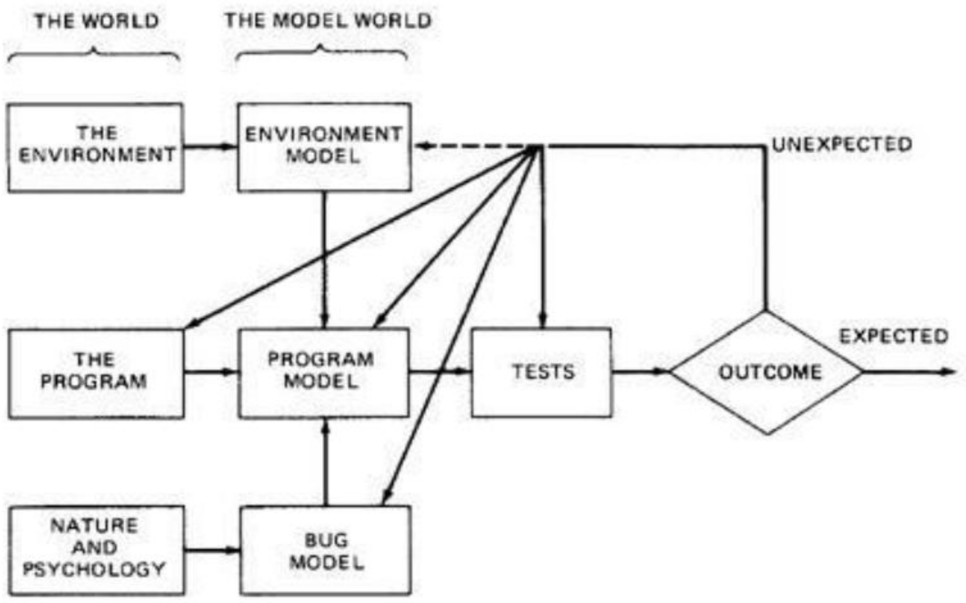


Figure 1.1: A Model for Testing

Above figure is a model of testing process. 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 in order to test it.

If simple model of the program doesn't explain the unexpected behavior, 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.

Optimistic notions about bugs:

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 affects only that component's behavior.

Control Bug Dominance: The belief those 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 Salvatore 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 is at code design.

Test s:

Tests are formal procedures, Inputs must be prepared, Outcomes should predict, 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.

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

Consequences of bugs: 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 behavior because of the bug is dehumanizing. *E.g.* Names are truncated or arbitrarily modified.

Disturbing: It refuses to handle legitimate (authorized / legal) transactions. The ATM won't 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 does not 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: What's 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.

TAXONOMY 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 Functionality 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 percentages 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 within 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.

Specification and Feature Bug Remedies:

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 tests and 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:

1. 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, spaghetti code, and worst of all, pachinko code.

One reason for control flow bugs is that this area is amenable (supportive) to theoretical 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 greater-than-or-equal 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 anomalies.

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 at least as common as bugs in code, but they are often treated as if they did not 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 Vs Static data:

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, which 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 subtlety 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.

Interface, integration, and system bugs:

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

1. 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 is 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 twofold: (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 toa 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, and 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 bugs 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.

FLOW GRAPHS AND PATH TESTING

BASICS OF PATH TESTING:

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.

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.
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.

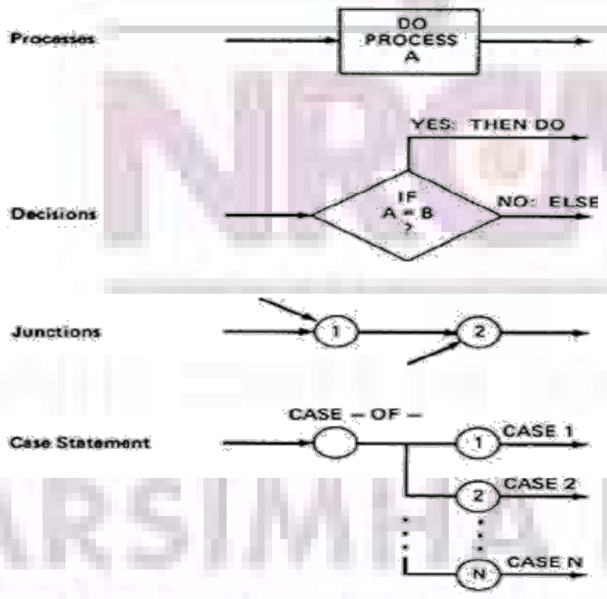


Figure 2.1: Flow graph Elements

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. o
In flow charts every part of the process block is drawn.

o The flowchart focuses on process steps, where as the flow graph focuses on control flow of the program.

o 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 Evolution:
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.

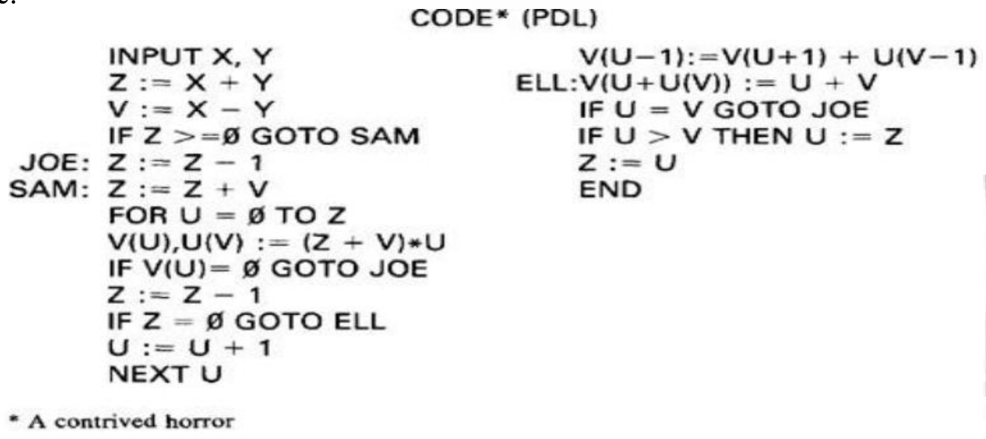
We don't need to know the specifics of the decisions, just the fact that there is a branch. o The specific target label names aren't important-just the fact that they exist. So we can

replace them by simple numbers.

o To understand this, we will go through an example (Figure 2.2) written in a FORTRAN like programming language called **Programming Design Language (PDL)**. The program's corresponding flowchart (Figure 2.3) and flowgraph (Figure 2.4) were also provided below for better understanding.

o The first step in translating the program to a flowchart is shown in Figure 2.3, 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 2.4 we merged the process steps and replaced them with the single process box.

We now have a control flow graph. But this representation is still too busy. We simplify the notation further to achieve Figure 2.5, 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 2.2: Program Example (PDL)

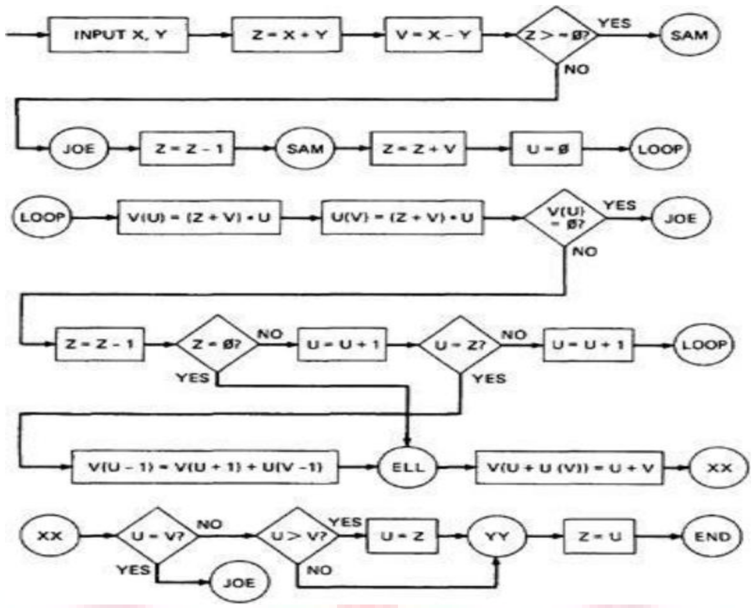


Figure 2.3: One-to-one flowchart for example program in Figure 2.2

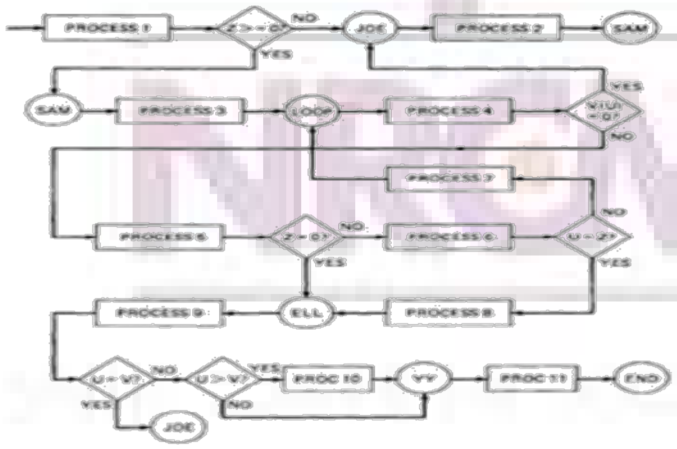


Figure 2.4: Control Flow graph for example in Figure 2.2

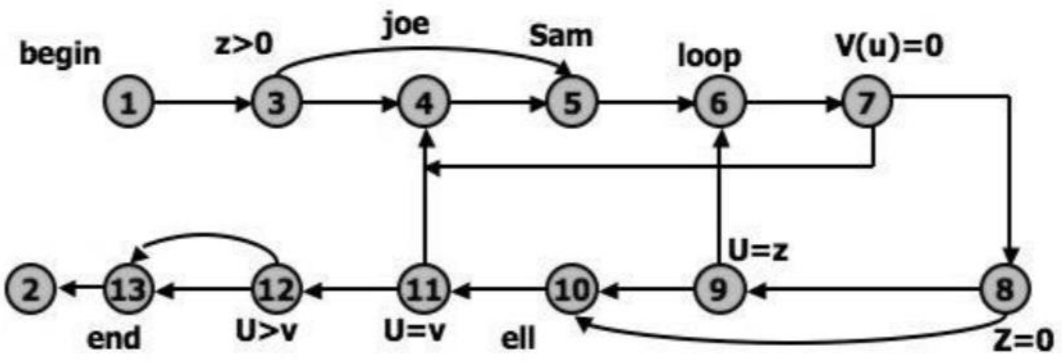


Figure 2.5: Simplified Flow graph Notation

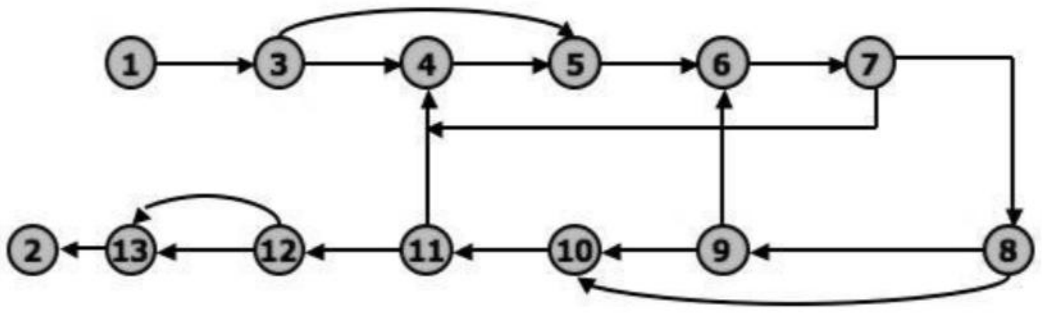


Figure 2.6: Even Simplified Flow graph Notation

The final transformation is shown in Figure 2.6, where we've dropped the node numbers to achieve an even simpler representation. The way to work with control flow graphs is to use the simplest possible representation - that is, no more information than you need to correlate back to the source program or PDL.

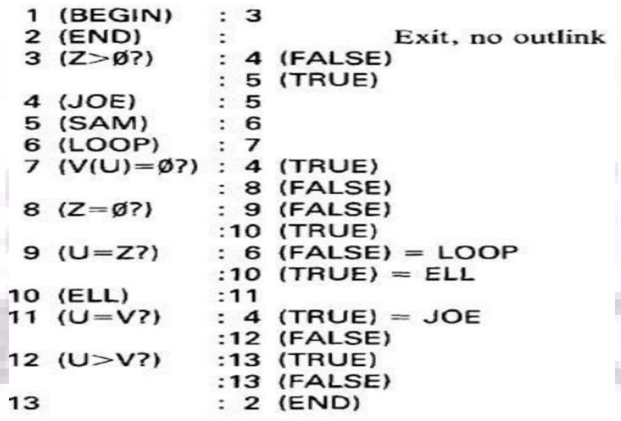
LINKED LIST REPRESENTATION:

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.

Linked List representation of Flow Graph:



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 2.7: Linked List Control Flow graph 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 can't always associate the parts of a program in a unique way with flow graph parts because many program structures, such as if-then-else constructs, consists of a combination of decisions, junctions, and processes. The translation from a flow graph element to a statement and vice versa is not always unique. (See Figure 2.8)

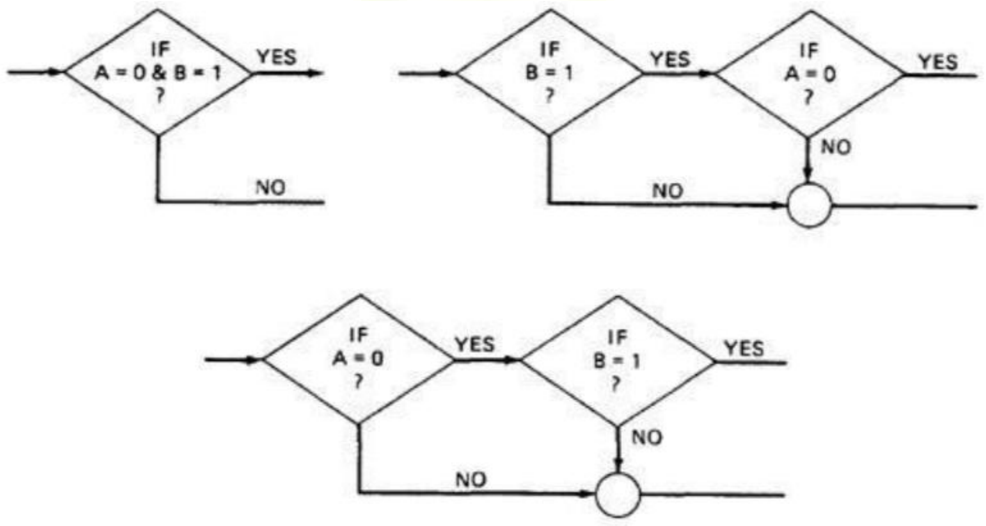


Figure 2.8: Alternative Flow graphs for

same logic (Statement "IF (A=0) AND

(B=1) THEN . . .").

An improper translation from flow graph 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 Handwritten by the programmer.

Automatically produced by a flowcharting program based on a mechanical analysis of the source code.
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.

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 samejunction, decision, or exit.
A path may go through several junctions, processes, or decisions, one or more times.

Paths consist 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. o 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. o The name of a path is the name of the nodes along the path.

FUNDAMENTAL PATH SELECTION CRITERIA:

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

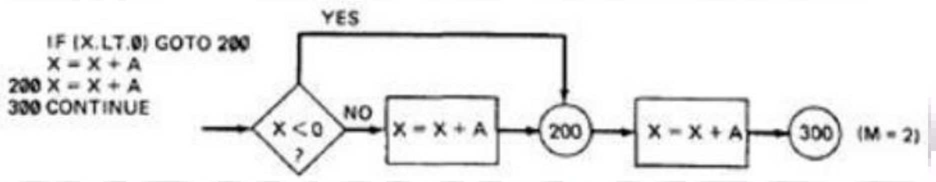
Every decision doubles the number of potential paths. And every loop multiplies the number of potential paths by the number of different iteration values possible for the loop.

Defining complete testing:

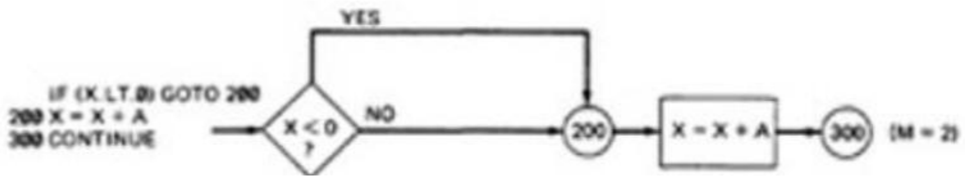
Exercise every path from entry to exit.
Exercise every statement or instruction at least once.
Exercise every branch and case statement, in each direction at least once.

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.

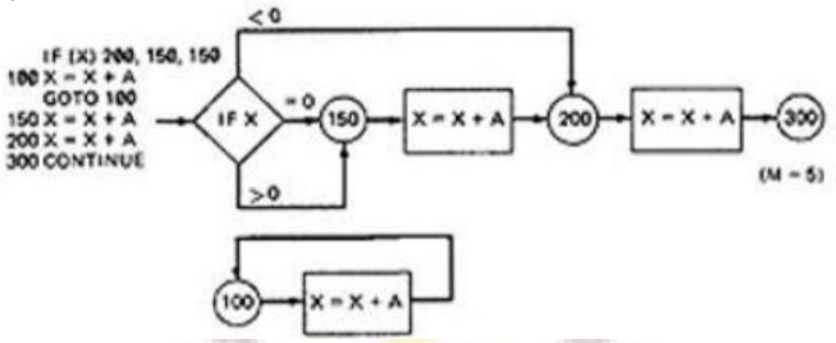
EXAMPLE*: Here is the correct version.*



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.

Only 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.

Path Testing (Pinf):

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.

Statement Testing (P1):

Execute all statements in the program at least once under some test. If we do enough tests to achieve this, we are said to have achieved 100% statement coverage.
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 cannot be accepted) and should be criminalized.

Branch Testing (P2):

Execute enough tests to assure that every branch alternative has been exercised at least once under some test.
If we do enough tests to achieve this prescription, then we have achieved 100% branch coverage.
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.

The question "why not use a judicious sampling of paths?, what is wrong with leaving some code, untested?" is ineffectual 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:

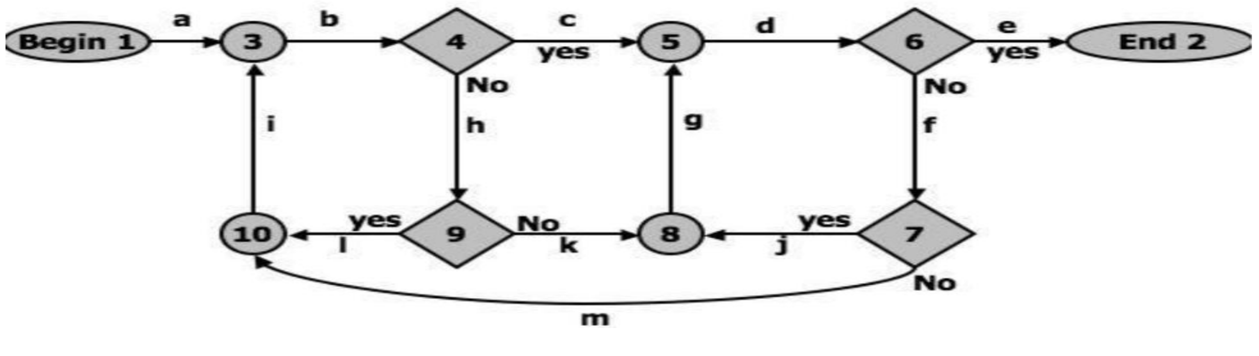


Figure 2.9: An example flow graph to explain path selection

Practical Suggestions in Path Testing:

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

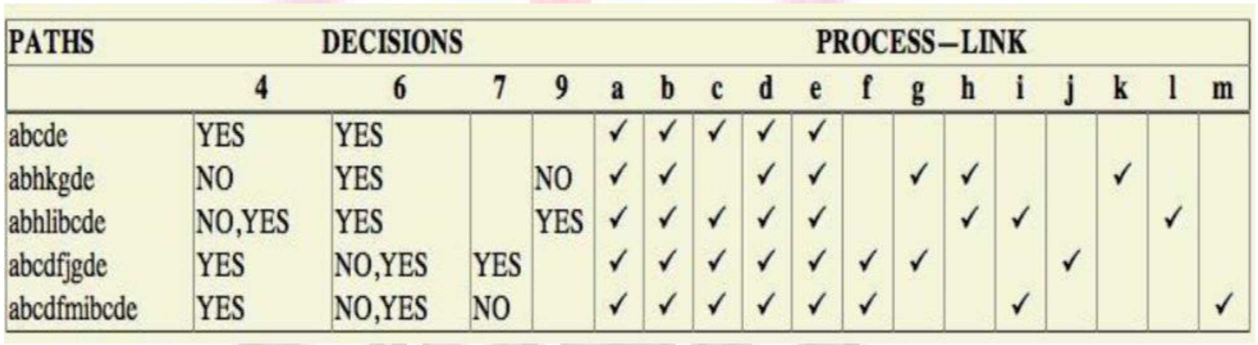
Make several copies - as many as you will need for coverage (C1+C2) and several more.

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

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

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

The above paths lead to the following table considering Figure 2.9:



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		✓	✓	✓	✓	✓	✓			✓				✓

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)

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.

Don't follow rules slavishly (blindly) - except for coverage.

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

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.

Could the loop-control variable be negative? Could it appear to specify a negative number of iterations? What happens to such a value?

One pass through the loop.

Two passes through the loop.

A typical number of iterations, unless covered by a previous test.
One less than the maximum number of iterations.
The maximum number of iterations.

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

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?
The minimum number of iterations.

One more than the minimum number of iterations.

Once, unless covered by a previous test.
Twice, unless covered by a previous test.
A typical value.
One less than the maximum value.
The maximum number of iterations.
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:

Start at the inner most loop. Set all the outer loops to their minimum values.

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.

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.

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

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.

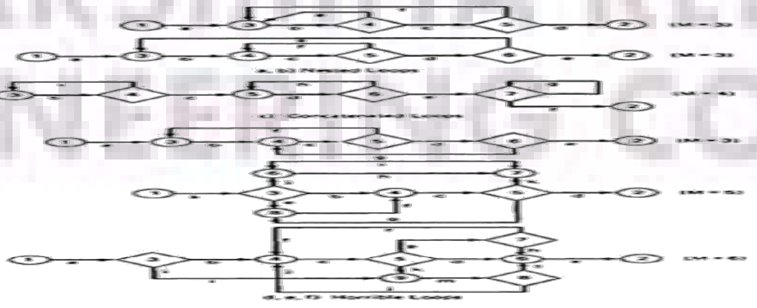


Figure 2.10: Example of Loop types

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.

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, ...

Z:=7@GOTOHEMB:Z:=-

7@GOTOHEMC: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:

X5>0	X6<0
X1 3X2+17	X1+3X2+17
= 0	= 0
X3 17	X3=17
X4-X1>=	X4-X1>=
X2	X2

- 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:

Assignment Blindness:

o 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.

o For Example:

orrect	uggy
..... > en ...	..... -Y > en ...

o 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:

orrect	uggy
Y = 2 then	Y = 2 then
.....	.....
X+Y > 3	X > 1
en ...	en ...

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

PATH SENSITIZING:

Review: achievable and unachievable paths:

We want to 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 un achievable.

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

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 un correlated, 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.

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.

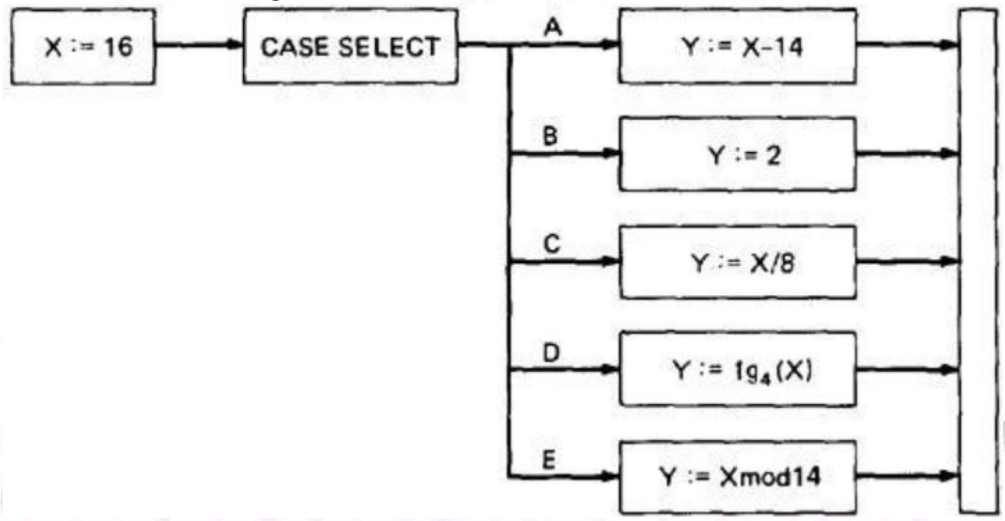


Figure 2.11: Coincidental Correctness

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:

o 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.

o Name every link by a lower case letter.

o Instrument the links so that the link's name is recorded when the link is executed.

o 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.

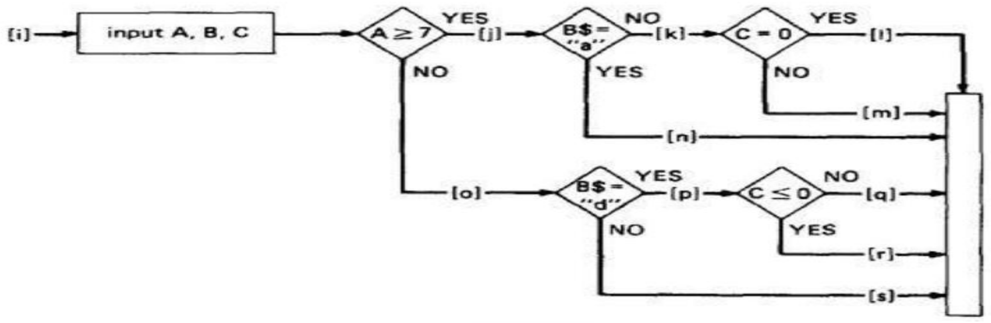


Figure 2.12: Single Link Marker Instrumentation

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.

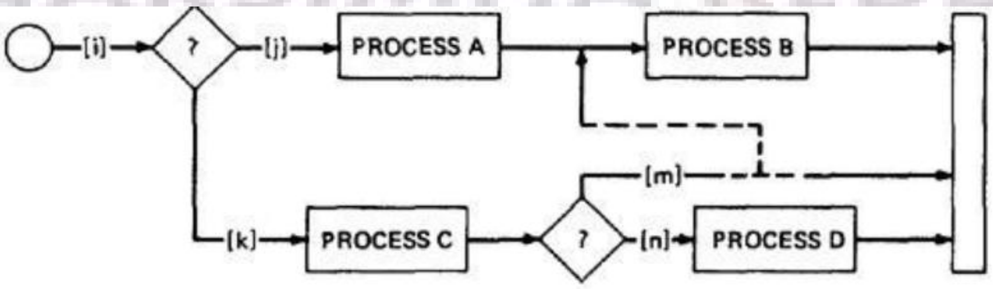


Figure 2.13: Why Single Link Markers aren't enough.

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.

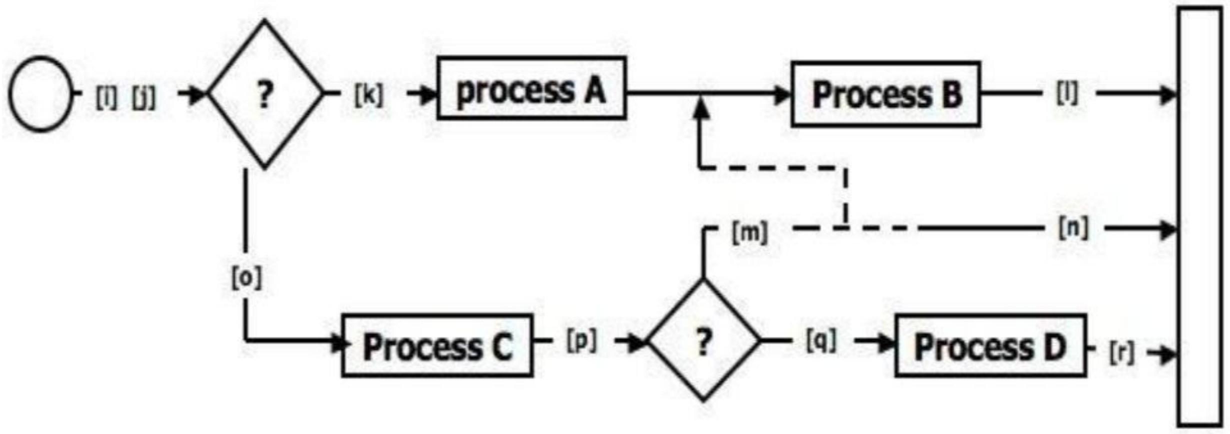


Figure 2.14: Double Link Marker Instrumentation

- **Link Counter:** 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.

UNIT-II

TRANSACTION FLOW TESTING AND DATA FLOW TESTING

Transaction Flow Testing:-transaction flows, transaction flow testing techniques. Dataflow testing:- Basics of dataflow testing, strategies in dataflow testing, application of dataflow testing. Domain Testing:-domains and paths, Nice & ugly domains, domain testing, domains and interfaces testing, domain and interface testing, domains and testability.

INTRODUCTION

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.

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:

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 are to the programmer.

The transaction flow graph is to create a behavioral model of the program that leads to functional testing.

The transaction flowgraph is a model of the structure of the system's behavior (functionality).

An example of a Transaction Flow is as follows:

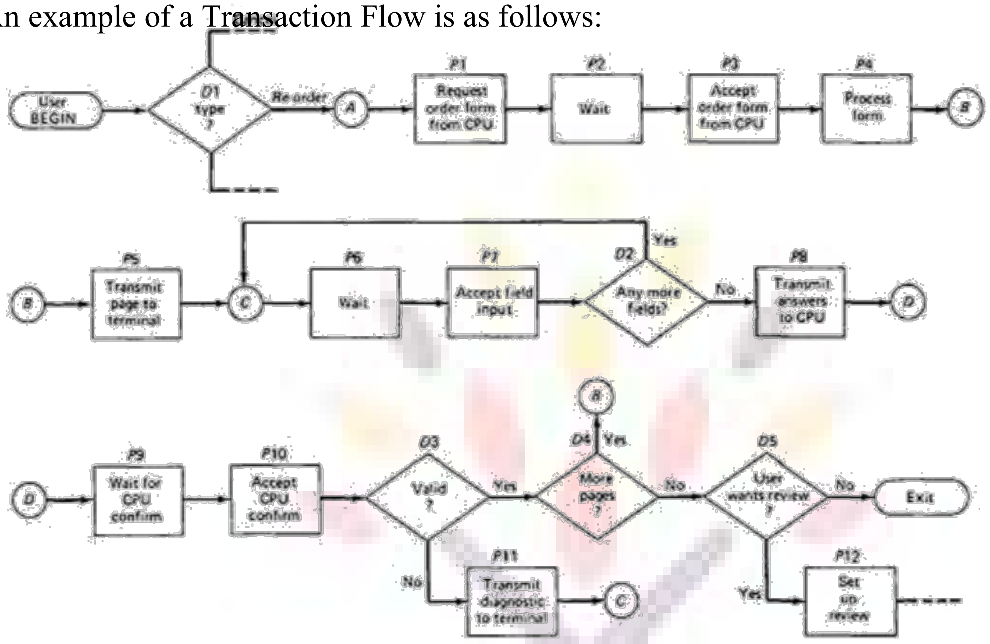


Figure 3.1: An Example of a Transaction Flow

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.

The flows are represented by relatively simple flowgraphs, many of which have a single straight-through path.

Loops are infrequent compared to control flowgraphs.

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.

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 a Mitosis.

Decision: Here the transaction will take one alternative or the other alternative but not both. (See Figure 3.2 (a))

Biosis: Here the incoming transaction gives birth to a new transaction, and both transaction continue on their separate paths, and the parent retains it identity. (See Figure 3.2 (b))

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

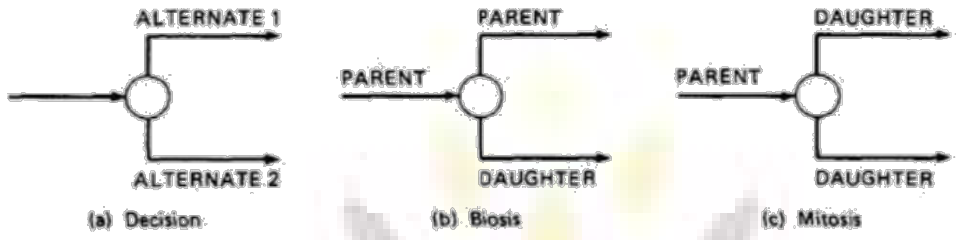


Figure 3.2: Nodes with multiple outlinks

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

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))

Absorption: In absorption case, the predator transaction absorbs prey transaction. The prey gone but the predator retains its identity. (See Figure 3.3 (b))

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))

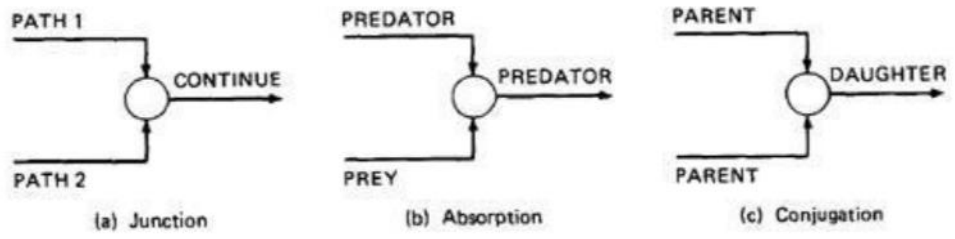


Figure 3.3: Transaction Flow Junctions and Mergers

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.

TRANSACTION FLOW TESTING TECHNIQUES:

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.

INSPECTIONS, REVIEWS AND WALKTHROUGHS:

Transaction flows are natural agenda for system reviews or inspections. o 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. o Design more test cases to validate all births and deaths.

o 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.

PATH SELECTION:

Select a set of covering paths (c1+c2) using the analogous criteria you used for structural path testing.

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.

PATH 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.

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.

BASICS OF DATA FLOW TESTING:

DATA FLOW TESTING:

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 sequences of events related to the status of data objects.

For example, pick enough paths to assure that every data object has beeninitialized prior to use or that all defined objects have been used for something.

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

DATA FLOW MACHINES:

There are two types of data flow machines with different architectures. (1) Von Neumann machines (2) Multi-instruction, multi-data machines (MIMD).

Von Neumann Machine Architecture:

Most computers today are von-neumann machines.

This architecture features interchangeable storage of instructions and data in the same memory units.

The Von Neumann machine Architecture executes one instruction at a time in the following, micro instruction sequence:

Fetch instruction from memory

Interpret instruction

Fetch operands
Process or Execute
Store result
Increment program counter

GOTO 1

Multi-instruction, Multi-data machines (MIMD) Architecture:

These machines can fetch several instructions and objects in parallel.

They can also do arithmetic and logical operations simultaneously on different data objects.

The decision of how to sequence them depends on the compiler.

BUG ASSUMPTION:

The bug assumption for data-flow testing strategies is that 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, or silly things are being done to data objects.

Also, if there is a control-flow problem, we expect it to have symptoms that can be detected by data-flow analysis.

Although we'll be doing data-flow testing, we won't be using data flow graphs as such. Rather, we'll use an ordinary control flow graph annotated to show what happens to the data objects of interest at the moment.

DATA FLOW GRAPHS:

The data flow graph is a graph consisting of nodes and directed links.
We will use a control graph to show what happens to data objects of interest at that moment.

Our objective is to expose deviations between the data flows we have and the data flows we want.

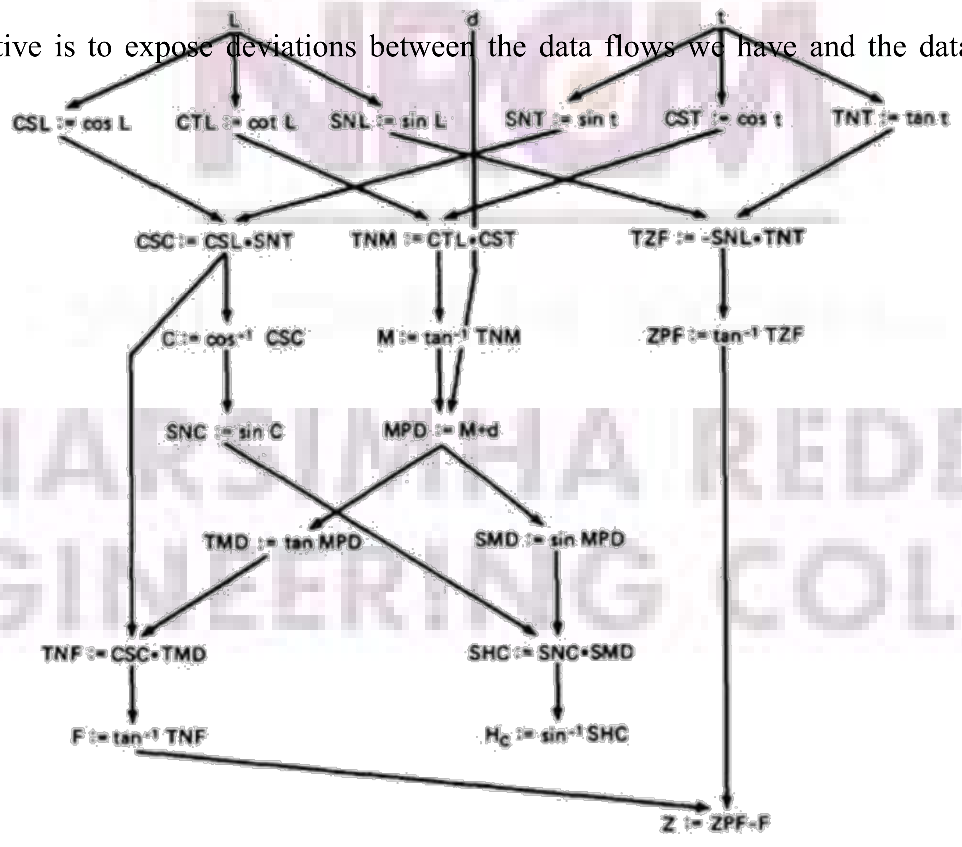


Figure 3.4: Example of a data flow graph

Data Object State and Usage:
Data Objects can be created, killed and used.

They can be used in two distinct ways: (1) In a Calculation (2) As a part of a Control Flow Predicate.

The following symbols denote these possibilities:
Defined: d - defined, created, initialized etc
Killed or undefined: k - killed, undefined, released etc

Usage: u - used for something (c - used in Calculations, p - used in a predicate)

1. Defined (d):

An object is defined explicitly when it appears in a data declaration.

Or implicitly when it appears on the left hand side of the assignment.

It is also to be used to mean that a file has been opened.
A dynamically allocated object has been allocated.

Something is pushed on to the stack.

A record written.

Killed or Undefined (k):
An object is killed on undefined when it is released or otherwise made unavailable.

When its contents are no longer known with certitude (with absolute certainty / perfectness).

Release of dynamically allocated objects back to the availability pool.

Return of records.

The old top of the stack after it is popped.

An assignment statement can kill and redefine immediately. For example, if A had been previously defined and we do a new assignment such as A : = 17, we have killed A's previous value and redefined A

Usage (u):

A variable is used for computation (c) when it appears on the right hand side of an assignment statement.

A file record is read or written.

It is used in a Predicate (p) when it appears directly in a predicate.

DATA FLOW ANOMALIES:

An anomaly is denoted by a two-character sequence of actions. For example, ku means that the object is killed and then used, where as dd means that the object is defined twice without an intervening usage.

What is an anomaly is depend on the application.

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

dd :- probably harmless but suspicious. Why define the object twice without an intervening usage?

dk :- probably a bug. Why define the object without using it?
du :- the normal case. The object is defined and then used.
kd :- normal situation. An object is killed and then redefined.
kk :- harmless but probably buggy. Did you want to be sure it was really killed?
ku :- a bug. the object doesnot exist.
ud :- usually not a bug because the language permits reassignment at almost any time.

uk :- normal situation.

uu :- normal situation.

In addition to the two letter situations, there are six single letter situations.We will use a leading dash to mean that nothing of interest (d,k,u) occurs prior to the action noted along the entry-exit path of interest.

A trailing dash to mean that nothing happens after the point of interest to the exit.

They possible anomalies are:

-k :- possibly anomalous because from the entrance to this point on the path, the variable had not been defined. We are killing a variable that does not exist.
-d :- okay. This is just the first definition along this path.
-u :- possibly anomalous. Not anomalous if the variable is global and has been previously defined.

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

d- :- possibly anomalous. The variable was defined and not used on this path. But this could be a global definition.

u- :- not anomalous. The variable was used but not killed on this path. Although this sequence is not anomalous, it signals a frequent kind of bug. If d and k mean dynamic storage allocation and return respectively, this could be an instance in which a dynamically allocated object was not returned to the pool after use.

DATA FLOW ANOMALY STATE GRAPH:

Data flow anomaly model prescribes that an object can be in one of four distinct states:

K :- undefined, previously killed, doesnot exist

D :- defined but not yet used for anything
U :- has been used for computation or in predicate
A :- anomalous

These capital letters (K, D, U, A) denote the state of the variable and should not be confused with the program action, denoted by lower case letters.

Unforgiving Data - Flow Anomaly Flow Graph: Unforgiving model, in which once a variable becomes anomalous it can never return to a state of grace.

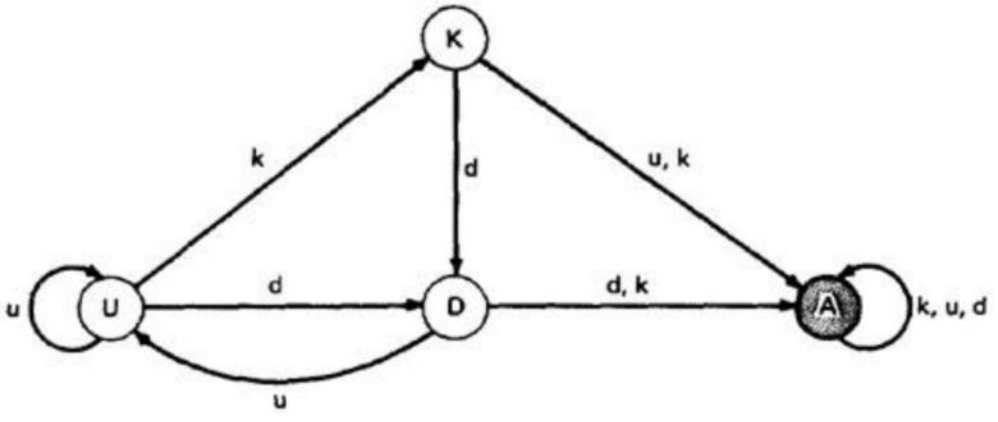

$$\begin{tikzcd}
 & K \arrow[d, "d"] \arrow[dr, "{u, k}"] & \\
U \arrow[ur, "k"] \arrow[r, "d"] \arrow[loop left, "u"] & D \arrow[r, "{d, k}"] \arrow[l, bend left, "u"] & A \arrow[loop right, "{k, u, d}"]
\end{tikzcd}$$

Figure 3.5: Unforgiving Data Flow Anomaly State Graph

Assume that the variable starts in the K state - that is, it has not been defined or does not exist. If an attempt is made to use it or to kill it (e.g., say that we're talking about opening, closing, and using files and that 'killing' means closing), the object's state becomes anomalous (state A) and, once it is anomalous, no action can return the variable to a working state.

If it is defined (d), it goes into the D, or defined but not yet used, state. If it has been defined (D) and redefined (d) or killed without use (k), it becomes anomalous, while usage (u) brings it to the U state. If in U, redefinition (d) brings it to D, u keeps it in U, and k kills it.

Forgiving Data - Flow Anomaly Flow Graph: Forgiving model is an alternate model where redemption (recover) from the anomalous state is possible

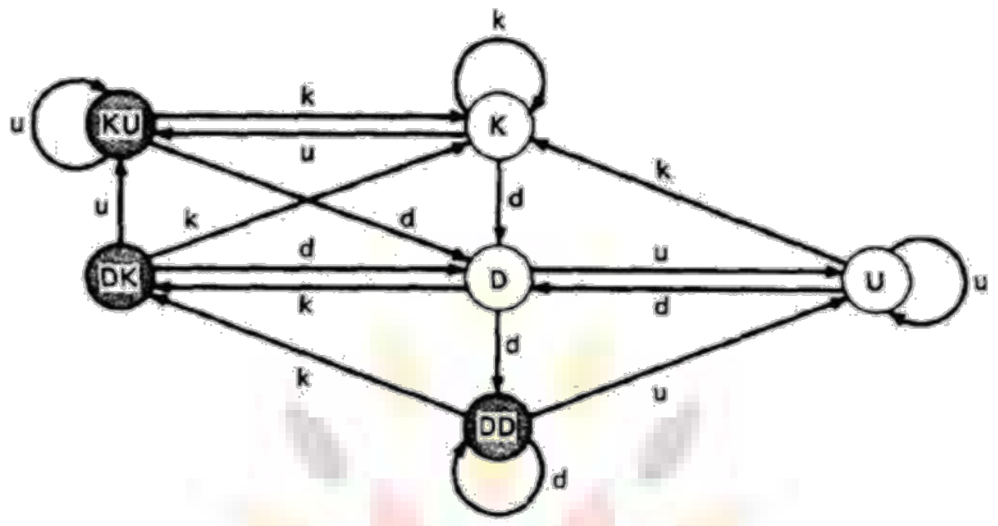


Figure 3.6: Forgiving Data Flow Anomaly State Graph

This graph has three normal and three anomalous states and he considers the kk sequence not to be anomalous. The difference between this state graph and Figure 3.5 is that redemption is possible. A proper action from any of the three anomalous states returns the variable to a useful working state.

The point of showing you this alternative anomaly state graph is to demonstrate that the specifics of an anomaly depends on such things as language, application, context, or even your frame of mind. In principle, you must create a new definition of data flow anomaly (e.g., a new state graph) in each situation. You must at least verify that the anomaly definition behind the theory or imbedded in a data flow anomaly test tool is appropriate to your situation.

STATIC Vs DYNAMIC ANOMALY DETECTION:

Static analysis is analysis done on source code without actually executing it. For example: source code syntax error detection is the static analysis result.

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. For example: a division by zero warning is the dynamic result.

If a problem, such as a data flow anomaly, can be detected by static analysis methods, then it doesn’t belongs in testing - it belongs in the language processor.

There is actually a lot more static analysis for data flow analysis for data flow anomalies going on in current language processors.

For example, language processors which force variable declarations can detect (-u) and (ku) anomalies.But still there are many things for which current notions of static analysis are INADEQUATE.

Why Static Analysis isn't enough? There are many things for which current notions of static analysis are inadequate. They are:

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.

Arrays: Arrays are problematic in that the array is defined or killed as a single object, but reference is to specific locations within the array. Array pointers are usually dynamically calculated, so there's no way to do a static analysis to validate the pointer value. In many languages, dynamically allocated arrays contain garbage unless explicitly initialized and therefore, -u anomalies are possible.

Records and Pointers: The array problem and the difficulty with pointers is a special case of multipart data structures. We have the same problem with records and the pointers to them. Also, in many applications we create files and their names dynamically and there's no way to determine, without execution, whether such objects are in the proper state on a given path or, for that matter, whether they exist at all.

Dynamic Subroutine and Function Names in a Call: subroutine or function name is a dynamic variable in a call. What is passed, or a combination of subroutine names and data objects, is constructed on a specific path. There's no way, without executing the path, to determine whether the call is correct or not.

False Anomalies: Anomalies are specific to paths. Even a "clear bug" such as ku may not be a bug if the path along which the anomaly exist is unachievable. Such "anomalies" are false anomalies. Unfortunately, the problem of determining whether a path is or is not achievable is unsolvable.

Recoverable Anomalies and Alternate State Graphs: What constitutes an anomaly depends on context, application, and semantics. How does the compiler know which model I have in mind? It can't because the definition of "anomaly" is not fundamental. The language processor must have a built-in anomaly definition with which you may or may not (with good reason) agree.

Concurrency, Interrupts, System Issues: As soon as we get away from the simple single-task uniprocessor environment and start thinking in terms of systems, most anomaly issues become vastly more complicated.

How often do we define or create data objects at an interrupt level so that they can be processed by a lower-priority routine? Interrupts can make the "correct" anomalous and the "anomalous" correct. True concurrency (as in an MIMD machine) and pseudo concurrency (as in multiprocessing) systems can do the same to us. Much of integration and system testing is aimed at detecting data-flow anomalies that cannot be detected in the context of a single routine.

Although static analysis methods have limits, they are worth using and a continuing trend in language processor design has been better static analysis methods, especially for data flow anomaly detection. That's good because it means there's less for us to do as testers and we have far too much to do as it is.

DATA FLOW MODEL:

The data flow model is based on the program's control flow graph - Don't confuse that with the program's data flow graph.

Here we annotate each link with symbols (for example, d, k, u, c, and p) or sequences of symbols (for example, dd, du, ddd) that denote the sequence of data operations on that link with respect to the variable of interest. Such annotations are called link weights.

The control flow graph structure is same for every variable: it is the weights that change.

Components of the model:

To every statement there is a node, whose name is unique. Every node has at least one outlink and at least one inlink except for exit nodes and entry nodes.

Exit nodes are dummy nodes placed at the outgoing arrowheads of exit statements (e.g., END, RETURN), to complete the graph. Similarly, entry nodes are dummy nodes placed at entry statements (e.g., BEGIN) for the same reason.

The outlink of simple statements (statements with only one outlink) are weighted by the proper sequence of data-flow actions for that statement. Note that the sequence can consist of more than one letter. For example, the assignment statement A:= A + B in most languages is weighted by cd or possibly ckd for variable A. Languages that permit multiple simultaneous assignments and/or compound statements can have anomalies within the statement. The sequence must correspond to the order in which the object code will be executed for that variable.

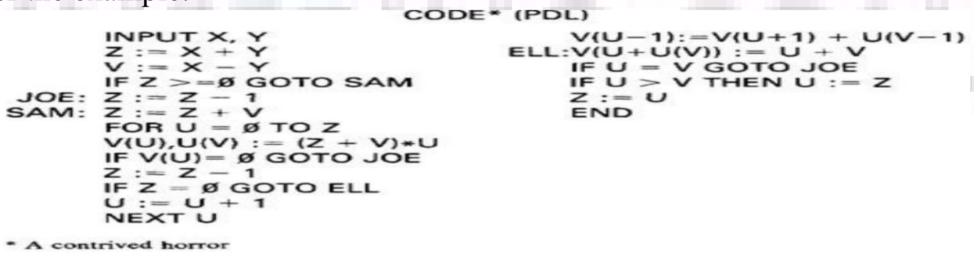
Predicate nodes (e.g., IF-THEN-ELSE, DO WHILE, CASE) are weighted with the p - use(s) on every outlink, appropriate to that outlink.

Every sequence of simple statements (e.g., a sequence of nodes with one inlink and one outlink) can be replaced by a pair of nodes that has, as weights on the link between them, the concatenation of link weights.

If there are several data-flow actions on a given link for a given variable, then the weight of the link is denoted by the sequence of actions on that link for that variable.

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.

Let us consider the example:



CODE* (PDL)

```
      INPUT X, Y
      Z := X + Y
      V := X - Y
      IF Z >=Ø GOTO SAM
JOE:  Z := Z - 1
SAM:  Z := Z + V
      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
      V(U-1):=V(U+1) + U(V-1)
ELL:  V(U+U(V)) := U + V
      IF U = V GOTO JOE
      IF U > V THEN U := Z
      Z := U
      END
```

* A contrived horror

Figure 3.7: Program Example (PDL)

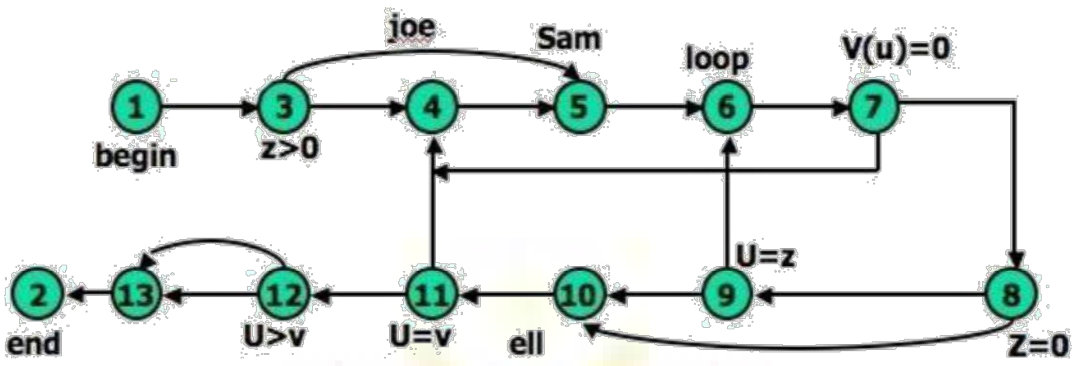


Figure 3.8: Unannotated flow graph for example program in Figure 3.7

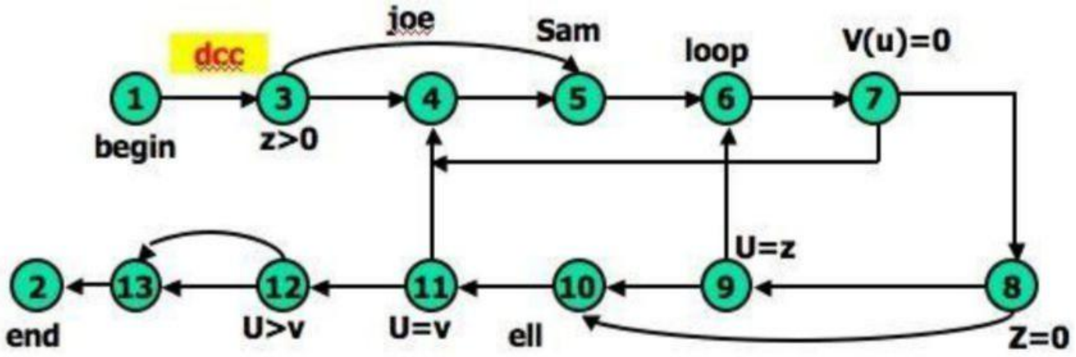


Figure 3.9: Control flow graph annotated for X and Y data flows.

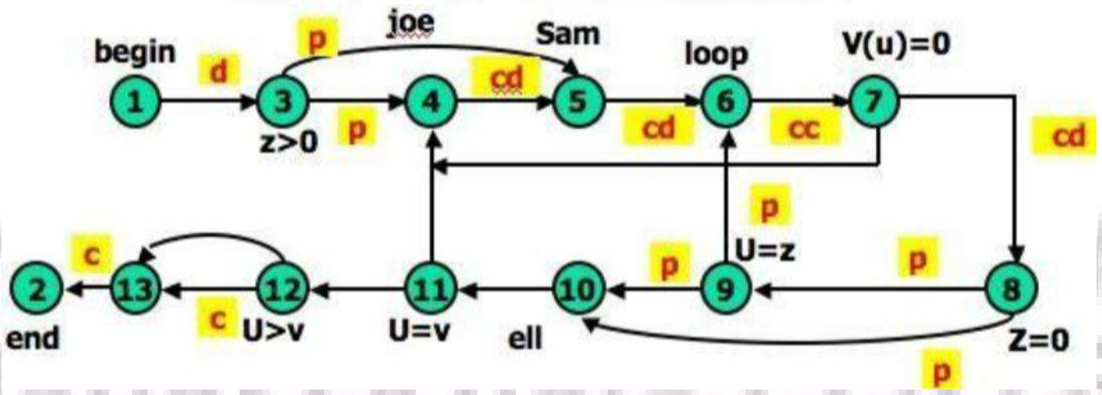


Figure 3.10: Control flow graph annotated for Z data flow.

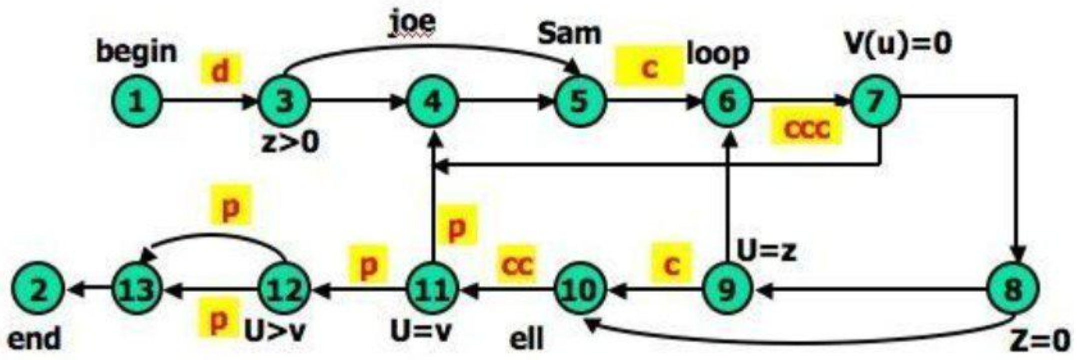


Figure 3.11: Control flow graph annotated for V data flow.

STRATEGIES OF DATA FLOW TESTING:

INTRODUCTION:

Data Flow Testing Strategies are structural strategies.
In contrast to the path-testing strategies, data-flow strategies take into account what happens to data objects on the links in addition to the raw connectivity of the graph.

In other words, data flow strategies require data-flow link weights (d,k,u,c,p).

Data Flow Testing Strategies are based on selecting test path segments (also called **sub paths**) that satisfy some characteristic of data flows for all data objects.
For example, all sub paths that contain a d (or u, k, du, dk).

A strategy X is **stronger** than another strategy Y if all test cases produced under Y are included in those produced under X - conversely for **weaker**.

TERMINOLOGY:
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. ll paths in

Figure

3.9 are definition clear because variables X and Y are defined only on the first link (1,3) and not thereafter. In Figure

3.10, we have a more complicated situation. The following path segments are definition-clear: (1,3,4), (1,3,5), (5,6,7,4), (7,8,9,6,7), (7,8,9,10), (7,8,10), (7,8,10,11). Subpath (1,3,4,5) is not definition-clear because the variable is defined on (1,3) and again on (4,5). For practice, try finding all the definition-clear subpaths for this routine (i.e., for all variables).

Loop-Free Path Segment is a path segment for which every node in it is visited atmost once. For Example, path (4,5,6,7,8,10) in Figure 3.10 is loop free, but path (10,11,4,5,6,7,8,10,11,12) is not because nodes 10 and 11 are each visited twice.

Simple path segment is a path segment in which at most one node is visited twice. For example, in Figure 3.10, (7,4,5,6,7) is a simple path segment. A simple path segment is either loop-free or if there is a loop, only one node is involved.

A **du path** from node i to k is a path segment such that if the last link has a computational use of X, then the path is simple and definition-clear; if the penultimate (last but one) node is j - that is, the path is (i,p,q,...,r,s,t,j,k) and link (j,k) has a predicate use - then the path from i to j is both loop-free and definition- clear.

STRATEGIES: The structural test strategies discussed below 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. Various types of data flow testing strategies in decreasing order of their effectiveness are:

All - du Paths (ADUP): 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 some test.

For variable X and Y:In Figure 3.9, because variables X and Y are used only on link (1,3), any test that starts at the entry satisfies this criterion (for variables X and Y, but not for all variables as required by the strategy).

For variable Z: The situation for variable Z (Figure 3.10) is more complicated because the variable is redefined in many places. For the definition on link (1,3) we must exercise paths that include subpaths (1,3,4) and (1,3,5). The definition on link (4,5) is covered by any path that includes (5,6), such as subpath (1,3,4,5,6, ...). The (5,6) definition requires paths that include subpaths (5,6,7,4) and (5,6,7,8).

For variable V: Variable V (Figure 3.11) is defined only once on link (1,3). Because V has a predicate use at node 12 and the subsequent path to the end must be forced for both directions at node 12, the all-du-paths strategy for this variable requires that we exercise all loop-free entry/exit paths and at least one path that includes the loop caused by (11,4).

Note that we must test paths that include both subpaths (3,4,5) and (3,5) even though neither of these has V definitions. They must be included because they provide alternate du paths to the V use on link (5,6). Although (7,4) is not used in the test set for variable V, it will be included in the test set that covers the predicate uses of array variable V() and U.

The all-du-paths strategy is a strong 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 definitions and uses of several different variables.

All Uses Startegy (AU): The all uses 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.

Just as we reduced our ambitions by stepping down from all paths (P) to branch coverage (C2), say, we can reduce the number of test cases by asking that the test set should include at least one path segment from every definition to every use that can be reached by that definition.

For variable V: In Figure 3.11, ADUP requires that we include subpaths (3,4,5) and (3,5) in some test because subsequent uses of V, such as on link (5,6), can be reached by either alternative. In AU either (3,4,5) or (3,5) can be used to start paths, but we don't have to use both. Similarly, we can skip the (8,10) link if we've included the (8,9,10) subpath.

Note the hole. We must include (8,9,10) in some test cases because that's the only way to reach the c use at link (9,10) - but suppose our bug for variable V is on link (8,10) after all? Find a covering set of paths under AU for Figure 3.11.

All p-uses/some c-uses strategy (APU+C) : For every variable and every definition of that variable, include at least one definition free path from the definition to every predicate use; if there are definitions of the variables that are not covered by the above prescription, then add computational use test cases as required to cover every definition.

For variable Z:In Figure 3.10, for APU+C we can select paths that all take the upper link (12,13) and therefore we do

not cover the c-use of Z: but that's okay according to the strategy's definition because every definition is covered.

Links (1,3), (4,5), (5,6), and (7,8) must be included because they contain definitions for variable

Z. Links (3,4), (3,5), (8,9), (8,10), (9,6), and (9,10) must be included because they contain predicate uses of Z. Find a covering set of test cases under APU+C for all variables inthis example - it only takes two tests.

For variable V:In Figure 3.11, APU+C is achieved for V by (1,3,5,6,7,8,10,11,4,5,6,7,8,10,11,12[upper], 13,2) and (1,3,5,6,7,8,10,11,12[lower], 13,2). Note

that the c-use at (9,10) need not be included under the APU+C criterion.

All c-uses/some p-uses strategy (ACU+P) : The all c-uses/some p-uses strategy (ACU+P) is to first ensure coverage by computational use cases and if any definition is not covered by the previously selected paths, add such predicate use cases as are needed to assure that every definition is included in some test.

For variable Z: In Figure 3.10, ACU+P coverage is achieved for Z by path (1,3,4,5,6,7,8,10, 11,12,13[lower], 2), but the predicate uses of several definitions are not covered. Specifically, the (1,3) definition is not covered for the (3,5) p-use, the (7,8) definition is not covered for the (8,9), (9,6) and (9, 10) p-uses.

The above examples imply that APU+C is stronger than branch coverage but ACU+P may be weaker than, or incomparable to, branch coverage.

All Definitions Strategy (AD) : The all definitions strategy asks only every definition of every variable be covered by

atleast one use of that variable, be that use a computational use or a predicate use.

For variable Z: Path (1,3,4,5,6,7,8, . . .) satisfies this criterion for variable Z, whereas any entry/exit path satisfies it for variable V.

From the definition of this strategy we would expect it to be weaker than both ACU+P and APU+C.

All Predicate Uses (APU), All Computational Uses (ACU) Strategies : The all predicate uses strategy is derived from 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. The all computational uses strategy is derived from ACU+P strategy by dropping the requirement that we include a p-use for the variable if there are no c-uses for the variable.

It is intuitively obvious that ACU should be weaker than ACU+P and that APU should be weaker than APU+C.

ORDERING THE STRATEGIES:

Figure 3.12compares path-flow and data-flow testing strategies. The arrows denote that the strategy at the arrow's tail is stronger than the strategy at the arrow's head

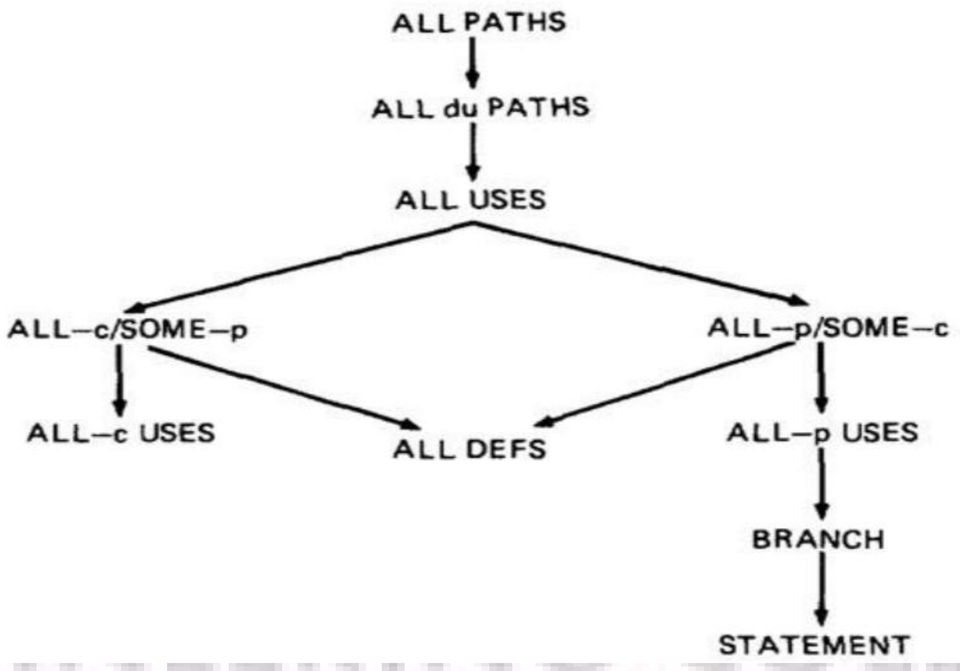


Figure 3.12: Relative Strength of Structural Test Strategies.

The right-hand side of this graph, along the path from "all paths" to "all statements" is the more interesting hierarchy for practical applications.

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

SLICING AND DICING:

A (static) program **slice** is a part of a program (e.g., a selected set of statements) defined with respect to a given variable X (where X is a simple variable or a data vector) and a statement i: it is the set of all statements that could (potentially, under static analysis) affect the value of X at statement i - where the influence of a faulty statement could result from an improper computational use or predicate use of some other variables at prior statements.

If X is incorrect at statement i, it follows that the bug must be in the program slice for X with respect to i

A program **dice** is a part of a 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).

The debugger first limits her scope to those prior statements that could have caused the faulty value at statement i (the slice) and then eliminates from further consideration those statements that testing has shown to be correct.

Debugging can be modeled as an iterative procedure in which slices are further refined by dicing, where the dicing information is obtained from ad hoc tests aimed primarily at eliminating possibilities. Debugging ends when the dice has been reduced to the one faulty statement.

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

DOMAIN TESTING

Domain Testing:-domains and paths, Nice & ugly domains, domain testing, domains and interfaces testing, domain and interface testing, domains and testability.

DOMAINS:

INTRODUCTION:

Domain: In mathematics, domain is a set of possible values of an independent variable or the variables of a function.

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

Domain testing can be based on specifications or equivalent implementation information.

If domain testing is based on specifications, it is a functional test technique.

If domain testing is based implementation details, it is a structural test technique. o For example, you're doing domain testing when you check extreme values of an input variable.

All inputs to a program can be considered as if they are numbers. For example, a character string can be treated as a number by concatenating bits and looking at them as if they were a binary integer. This is the view in domain testing, which is why this strategy has a mathematical flavor.

THE MODEL: The following figure is a schematic representation of domain testing.

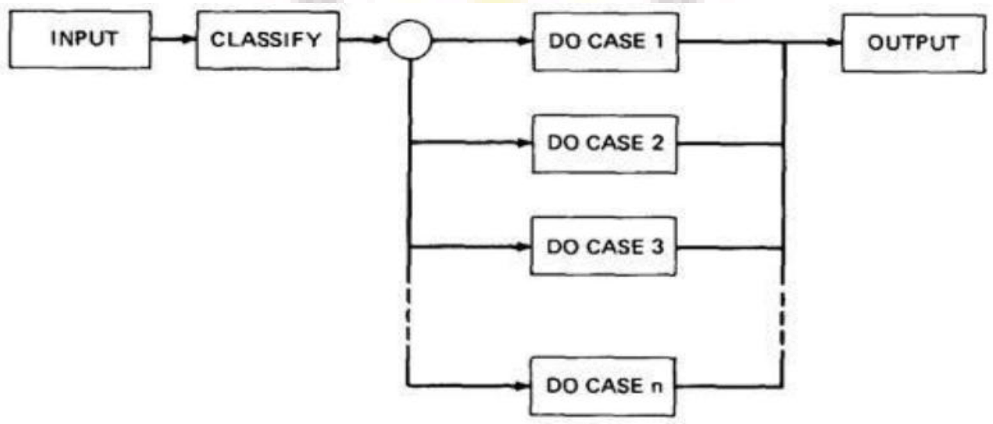


Figure 4.1: Schematic Representation of Domain Testing.

Before doing whatever it does, a routine must classify the input and set it moving on the right path.

An invalid input (e.g., value too big) is just a special processing case called 'reject'.

The input then passes to a hypothetical subroutine rather than on calculations. o In domain testing, we focus on the classification aspect of the routine rather than on the calculations. Structural knowledge is not needed for this model - only a consistent, complete specification of input values for each case.

We can infer that for each case there must be at least one path to process that case.

A DOMAIN IS A SET:

An input domain is a set.
If the source language supports set definitions (E.g. PASCAL set types and C enumerated types) less testing is needed because the compiler does much of it for us.

Domain testing does not work well with arbitrary discrete sets of data objects.
Domain for a loop-free program corresponds to a set of numbers defined over the input vector.

DOMAINS, PATHS AND PREDICATES:

In domain testing, predicates are assumed to be interpreted in terms of input vector variables.

If domain testing is applied to structure, then predicate interpretation must be based on actual paths through the routine - that is, based on the implementation control flow graph.

Conversely, if domain testing is applied to specifications, interpretation is based on a specified data flow graph for the routine; but usually, as is the nature of specifications, no interpretation is needed because the domains are specified directly.

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.

Domains are defined their boundaries. Domain boundaries are also where most domain bugs occur.

For every boundary there is at least one predicate that specifies what numbers belong to the domain and what numbers don't.
For example, 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).

o A domain may have one or more boundaries - no matter how many variables

define it. For example, if the predicate is x2 + y2 < 16, the domain is the inside of a circle of radius 4 about the origin. Similarly, we could define a spherical domain with one boundary but in three variables.

Domains are usually defined by many boundary segments and therefore by many predicates. i.e. the set of interpreted predicates traversed on that path (i.e., the path's predicate expression) defines the domain's boundaries.

A DOMAIN CLOSURE:

A domain boundary is **closed** with respect to a domain if the points on the boundary belong to the domain.

If the boundary points belong to some other domain, the boundary is said to be **open**.
Figure 4.2 shows three situations for a one-dimensional domain - i.e., a domain defined over one input variable; call it x
The importance of domain closure is that incorrect closure bugs are frequent domain bugs. For example, x >= 0 when x > 0 was intended

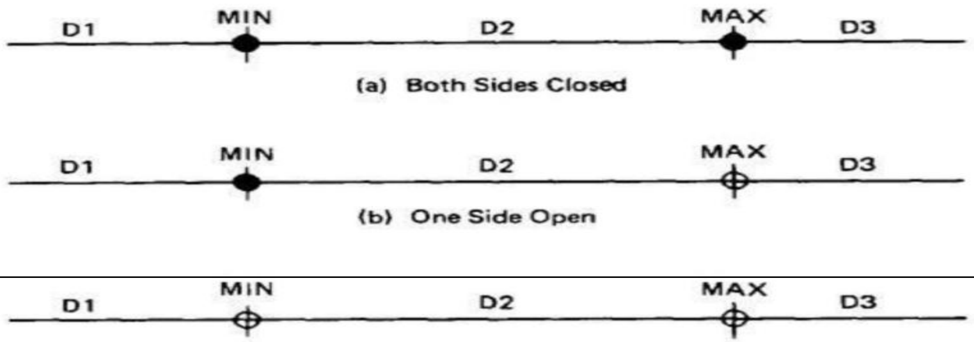


Figure 4.2: Open and Closed Domains.

DOMAIN DIMENSIONALITY:

Every input variable adds one dimension to the domain.
One variable defines domains on a number line.
Two variables define planardomains.
Three variables define solid domains.
Every new predicate slices through previously defined domains and cuts them in half.
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, volumes by planes, lines and points and n-spaces by hyperplanes.

BUG ASSUMPTION:
The bug assumption for the domain testing is that processing is okay but the domain definition is wrong.An incorrectly implemented domain means that boundaries are wrong, which may in turn mean that control flow predicates are wrong.

o Many different bugs can result in domain errors. Some of them are:

Domain Errors:

Double Zero Representation: In computers or Languages that have a distinct positive and negative zero, boundary errors for negative zero are common.

Floating point zero check: A floating point number can equal zero only if the previous definition of that number set it to zero or if it is subtracted from itself or multiplied by zero. So the floating point zero check to be done against an epsilon value.
Contradictory domains: An implemented domain can never be ambiguous or contradictory, but a specified domain can. A contradictory domain specification means that at least two supposedly distinct domains overlap.

Ambiguous domains: Ambiguous domains means that union of the domains is incomplete. That is there are missing domains or holes in the specified domains. Not specifying what happens to points on the domain boundary is a common ambiguity.

Over specified Domains: his domain can be overloaded with so many conditions that the result is a null domain. Another way to put it is to say that the domain's path is unachievable.

Boundary Errors: Errors caused in and around the boundary of adomain. Example, boundary closure bug, shifted, tilted, missing, extra boundary.

Closure Reversal: A common bug. The predicate is defined in terms of >=. The programmer chooses to implement the logical complement and incorrectly uses <= for the new predicate; i.e., x >= 0 is incorrectly negated as x <= 0, thereby shifting boundary values to adjacent domains.

Faulty Logic: Compound predicates (especially) are subject to faulty logic transformations and improper simplification. If the predicates define domain boundaries, all kinds of domain bugs can result from faulty logic manipulations.

RESTRICTIONS TO DOMAIN TESTING: Domain testing has restrictions, as do other testing techniques. Some of them include:

Co-incidental Correctness: Domain testing isn't good at finding bugs for which the outcome is correct for the wrong reasons. If we're plagued by coincidental correctness we may misjudge an incorrect boundary. Note that this implies weakness for domain testing when dealing with routines that have binary outcomes (i.e., TRUE/FALSE)

Representative Outcome: Domain testing is an example of **partition testing**. Partition-testing strategies divide the program's input space into domains such that all inputs within a domain are equivalent (not equal, but equivalent) in the sense that any input represents all inputs in that domain.

If the selected input is shown to be correct by a test, then processing is presumed correct, and therefore all inputs within that domain are expected (perhaps unjustifiably) to be correct. Most test techniques, functional or structural, fall under partition testing and therefore make this representative outcome assumption. For example, x^2 and 2^x are equal for $x = 2$, but the functions are different. The functional differences between adjacent domains are usually simple, such as $x + 7$ versus $x + 9$, rather than x^2 versus 2^x.

Simple Domain Boundaries and Compound Predicates: Compound predicates in which each part of the predicate specifies a different boundary are not a problem: for example, x

>= 0 AND x < 17, just specifies two domain boundaries by one compound predicate. As

an example of a compound predicate that specifies one boundary, consider: x = 0 AND y

>= 7 AND y <= 14. This predicate specifies one boundary equation (x = 0) but alternates closure, putting it in one or the other domain depending on whether y < 7 or y > 14. Treat compound predicates with respect because they're more complicated than they seem.

Functional Homogeneity of Bugs: Whatever the bug is, it will not change the functional form of the boundary predicate. For example, if the predicate is ax >= b, the bug will be in the value of a or b but it will not change the predicate to ax >= b, say.

Linear Vector Space: Most papers on domain testing, assume linear boundaries - not a bad assumption because in practice most boundary predicates are linear.

Loop Free Software: Loops are problematic for domain testing. The trouble with loops is that each iteration can result in a different predicate expression (after interpretation), which means a possible domain boundary change.

NICE AND UGLY DOMAINS:

NICE DOMAINS:

Where do these domains come from?

Domains are and will be defined by an imperfect iterative process aimed at achieving (user, buyer, voter) satisfaction. Implemented domains can't be incomplete or inconsistent. Every input will be processed (rejection is a process), possibly forever. Inconsistent domains will be made consistent.

o Conversely, specified domains can be incomplete and/or inconsistent. Incomplete in this context means that there are input vectors for which no path is specified, and inconsistent means that there are at least two contradictory specifications over the same segment of the input space.

Some important properties of nice domains are: **Linear, Complete, Systematic,**

And Orthogonal, Consistently closed, Convex and simply connected.
To the extent that domains have these properties domain testing is easy as testing gets.

o The bug frequency is lesser for nice domain than for ugly domains.

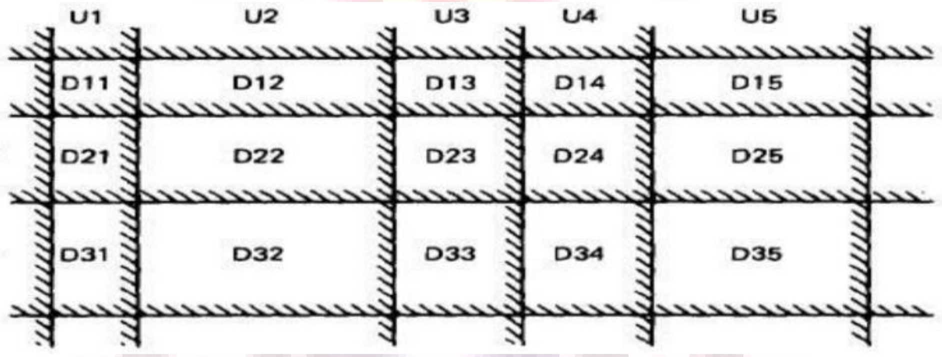


Figure 4.3: Nice Two-Dimensional Domains.

LINEAR AND NON LINEAR BOUNDARIES:

Nice domain boundaries are defined by linear inequalities or equations.
The impact on testing stems from the fact that it takes only two points to determine a straight line and three points to determine a plane and in general n+ 1 point to determine an n-dimensional hyper plane.

In practice more than 99.99% of all boundary predicates are either linear or can be linearized by simple variable transformations.

COMPLETE BOUNDARIES:
Nice domain boundaries are complete in that they span the number space from plus to minus infinity in all dimensions.

Figure 4.4 shows some incomplete boundaries. Boundaries A and E have gaps.

$$\begin{array}{ll} f_1(X) >= k_1 \text{ or } & f_1(X) >= g(1,c) \\ f_1(X) >= k_2 & f_2(X) >= g(2,c) \\ \ldots\ldots\ldots\ldots & \ldots\ldots\ldots\ldots \\ f_i(X) >= k_i & f_i(X) >= g(i,c) \end{array}$$

Such boundaries can come about because the path that hypothetically corresponds to them is unachievable, because inputs are constrained in such a way that such values can't exist, because of compound predicates that define a single boundary, or because redundant predicates convert such boundary values into a null set.

The advantage of complete boundaries is that one set of tests is needed to confirm the boundary no matter how many domains it bounds.

If the boundary is chopped up and has holes in it, then every segment of that boundary must be tested for every domain it bounds.

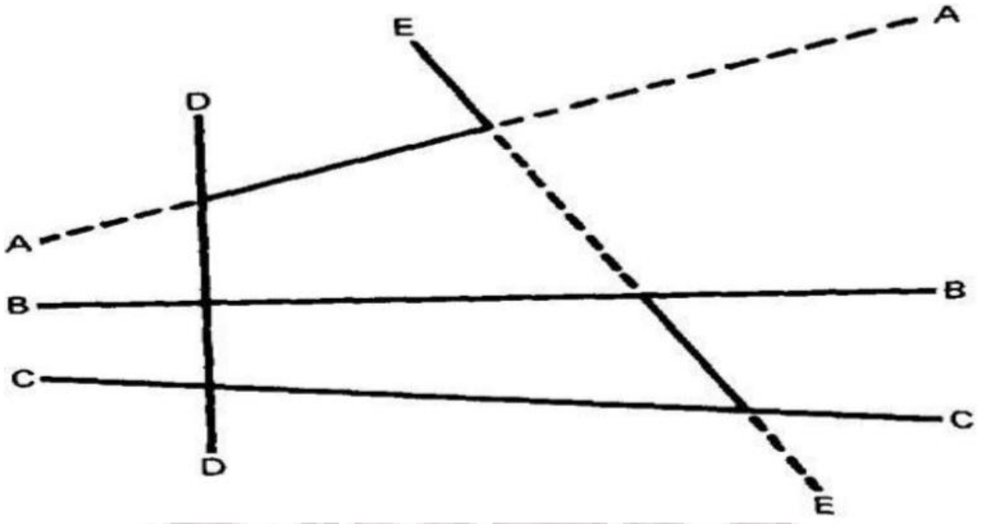


Figure 4.4: Incomplete Domain Boundaries.

SYSTEMATIC BOUNDARIES:

Systematic boundary means that boundary inequalities related by a simple function such as a constant.

In Figure 4.3 for example, the domain boundaries for u and v differ only by a constant.
where *fi* is an arbitrary linear function, X is the input vector, *ki* and *c* are constants, and *g(i,c)* is a decent function over *i* and *c* that yields a constant, such as *k* + *ic*.

The first example is a set of parallel lines, and the second example is a set of systematically (e.g., equally) spaced parallel lines (such as the spokes of a wheel, if equally spaced in angles, systematic).

If the boundaries are systematic and if you have one tied down and generate tests for it, the tests for the rest of the boundaries in that set can be automatically generated.

ORTHOGONAL BOUNDARIES:

Two boundary sets U and V (See Figure 4.3) are said to be orthogonal if every inequality in V is perpendicular to every inequality in U.

If two boundary sets are orthogonal, then they can be tested independently

In Figure 4.3 we have six boundaries in U and four in V. We can confirm the boundary properties in a number of tests proportional to 6 + 4 = 10 (O(n)). If we tilt the boundaries to get Figure 4.5, we must now test the intersections. We've gone from a linear number of cases to a quadratic: from O(n) to O(n^2).

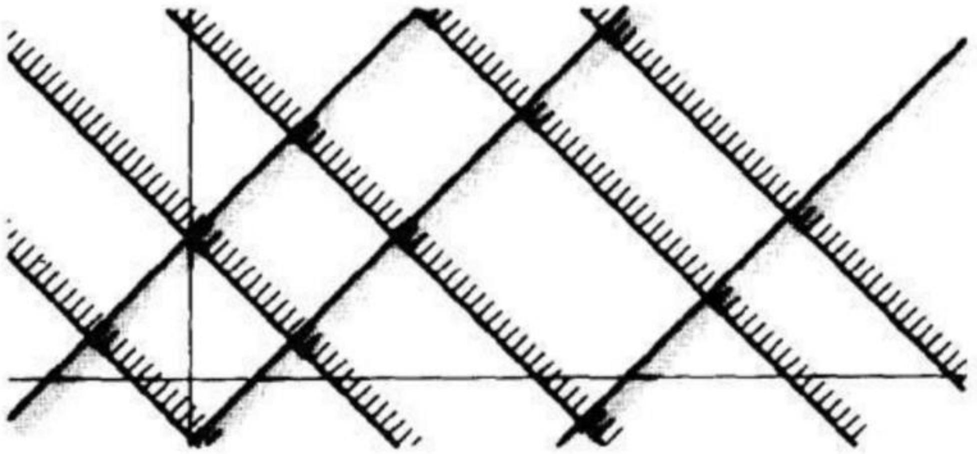


Figure 4.5: Tilted Boundaries.

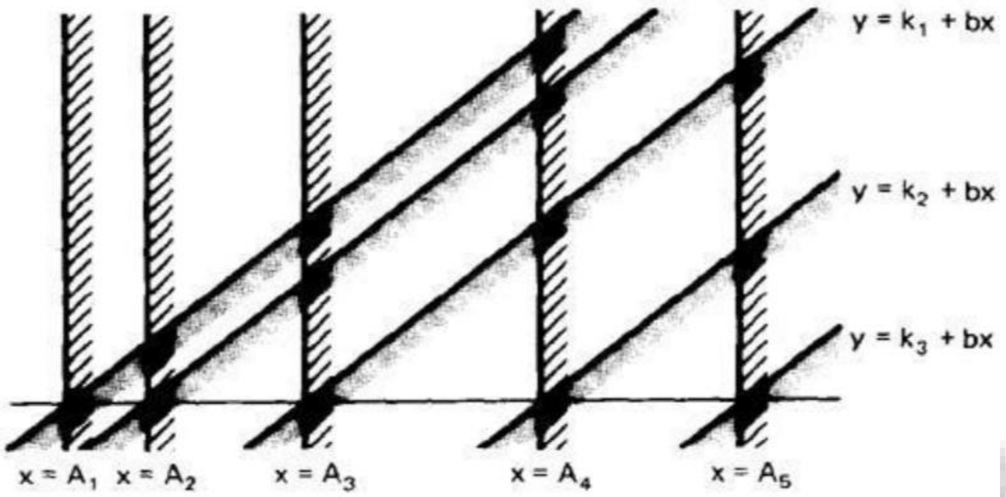


Figure 4.6: Linear, Non-orthogonal Domain Boundaries.

Actually, there are two different but related orthogonality conditions. Sets of boundaries can be orthogonal to one another but not orthogonal to the coordinate axes (condition 1), or boundaries can be orthogonal to the coordinate axes (condition 2).

CLOSURE CONSISTENCY:

Figure 4.6 shows another desirable domain property: boundary closures are consistent and systematic.

The shaded areas on the boundary denote that the boundary belongs to the domain in which the shading lies - e.g., the boundary lines belong to the domains on the right.

Consistent closure means that there is a simple pattern to the closures - for example, using the same relational operator for all boundaries of a set of parallel boundaries.

CONVEX:
A geometric figure (in any number of dimensions) is convex if you can take two arbitrary points on any two different boundaries, join them by a line and all

points on that line lie within the figure.

Nice domains are convex; dirty domains aren't.
You can smell a suspected concavity when you see phrases such as: ". . . except if

. . .," "However . . .," ". . . but not. . . ." In programming, it's often the buts in the specification that kill you.

SIMPLY CONNECTED:
Nice domains are simply connected; that is, they are in one piece rather than pieces all over the place interspersed with other domains.

Simple connectivity is a weaker requirement than convexity; if a domain is convex it is simply connected, but not vice versa.

Consider domain boundaries defined by a compound predicate of the (Boolean) form ABC. Say that the input space is divided into two domains, one defined by
ABC and, therefore, the other defined by its negation.

For example, suppose we define valid numbers as those lying between 10 and 17 inclusive. The invalid numbers are the disconnected domain consisting of numbers less than 10 and greater than 17.
Simple connectivity, especially for default cases, may be impossible.

UGLY DOMAINS:
Some domains are born ugly and some are uglified by bad specifications.
Every simplification of ugly domains by programmers can be either good orbad. o Programmers in search of nice solutions will "simplify" essential complexity out of existence. Testers in search of brilliant insights will be blind to essential complexity and therefore miss important cases.

o If the ugliness results from bad specifications and the programmer's simplification is harmless, then the programmer has made ugly good.

But if the domain's complexity is essential (e.g., the income tax code), such "simplifications" constitute bugs.

Nonlinear boundaries are so rare in ordinary programming that there's no information on how programmers might "correct" such boundaries if they're essential.

AMBIGUITIES AND CONTRADICTIONS:

Domain ambiguities are holes in the input space.
The holes may lie within the domains or in cracks between domains.

Two kinds of contradictions are possible: overlapped domain specifications and overlapped closure specifications

Figure 4.7c shows overlapped domains and Figure 4.7d shows dual closure assignment.

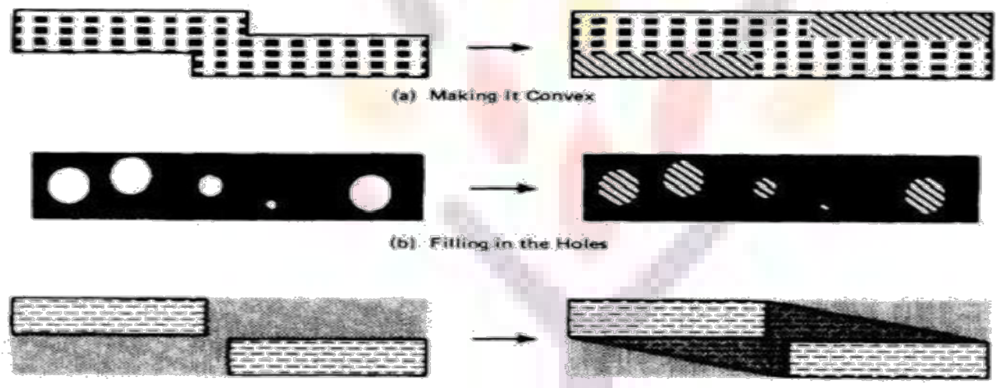


Figure 4.7: Domain Ambiguities and Contradictions.

SIMPLIFYING THE TOPOLOGY:

The programmer's and tester's reaction to complex domains is the same - simplify o There are three generic cases: **concavities**, **holes** and **disconnected pieces**.
o Programmers introduce bugs and testers misdesign test cases by: smoothing out concavities (Figure 4.8a), filling in holes (Figure 4.8b), and joining disconnected pieces (Figure 4.8c).

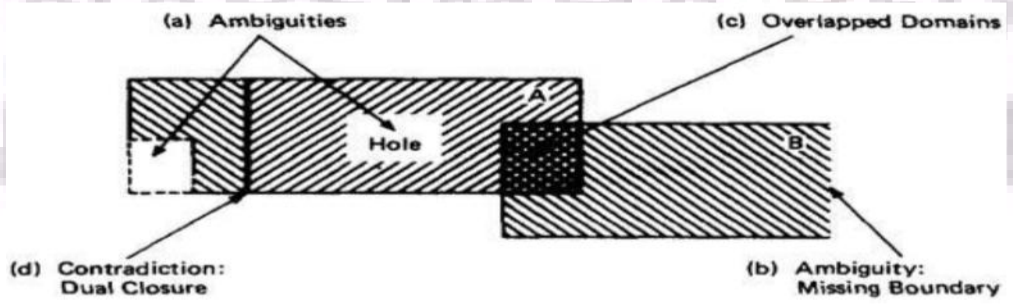

(a) Ambiguities
(c) Overlapped Domains
A
Hole
B
(d) Contradiction: Dual Closure
(b) Ambiguity: Missing Boundary

Figure 4.8: Simplifying the topology.

RECTIFYING BOUNDARY CLOSURES:

If domain boundaries are parallel but have closures that go every which way (left, right, left . . .) the natural reaction is to make closures go the same way (see Figure 4.9).

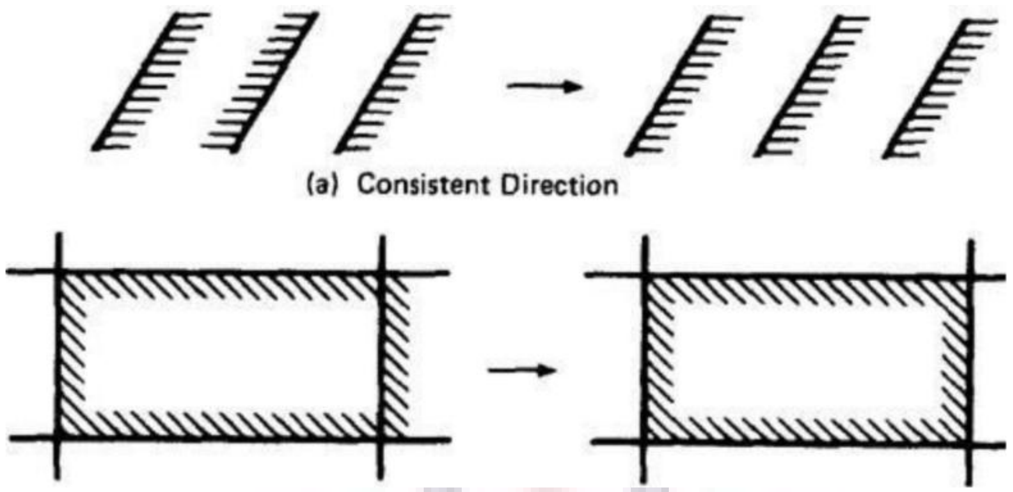


Figure 4.9: Forcing Closure Consistency.

DOMAIN TESTING:

DOMAIN TESTING STRATEGY: The domain-testing strategy is simple, although possibly tedious (slow).

o Domains are defined by their boundaries; therefore, domain testing concentrates test points on or near boundaries.

o Classify what can go wrong with boundaries, then define a test strategy for each case. Pick enough points to test for all recognized kinds of boundary errors.

o 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.

o Run the tests and by posttest analysis (the tedious part) determine if anyboundaries are faulty and if so, how.

Run enough tests to verify every boundary of everydomain.

DOMAIN BUGS AND HOW TO TEST FOR THEM:

An **interior point** (Figure 4.10) is a point in the domain such that all points within an arbitrarily small distance (called an epsilon neighborhood) are also in the domain.

A **boundary point** is one such that within an 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.

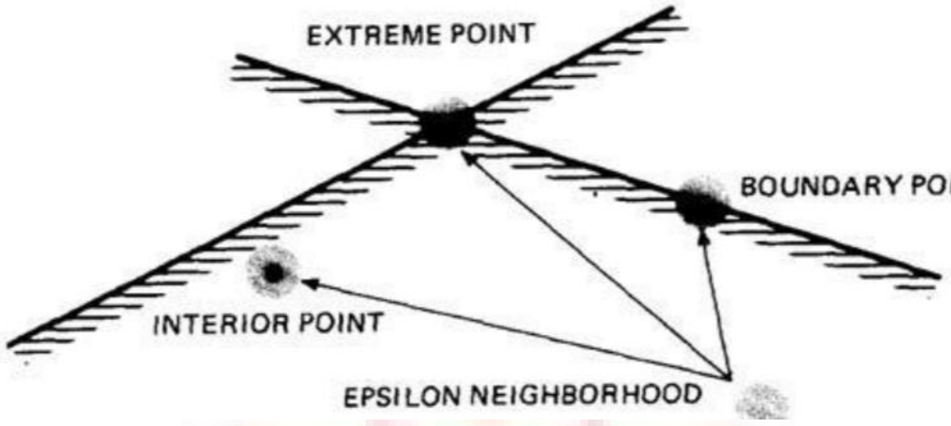


Figure 4.10: Interior, Boundary and Extreme points.

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.

If the boundary is open, an off point is a point near the boundary but in the domain being tested; see Figure 4.11. You can remember this by the acronym COOOOI: Closed Off Outside, Open Off Inside.

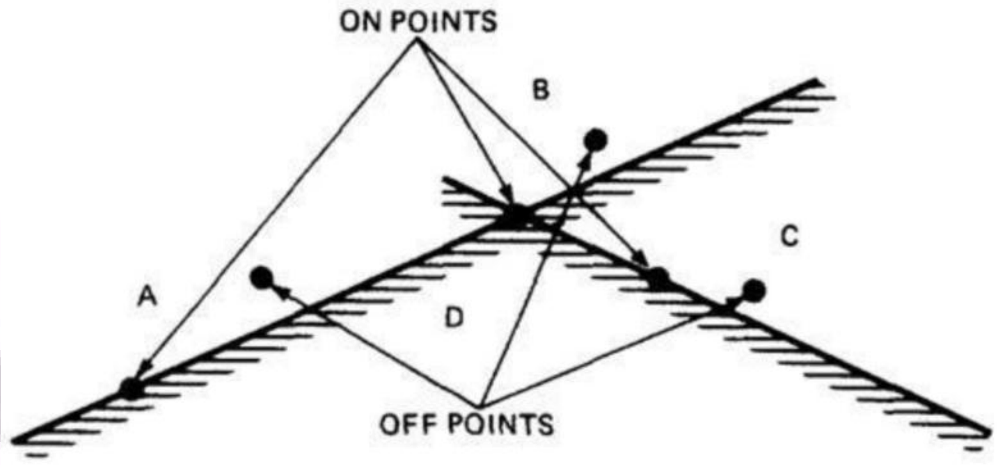


Figure 4.11: On points and Off points.

=Figure 4.12 shows generic domain bugs: closure bug, shifted boundaries, tilted boundaries, extra boundary, missing boundary.

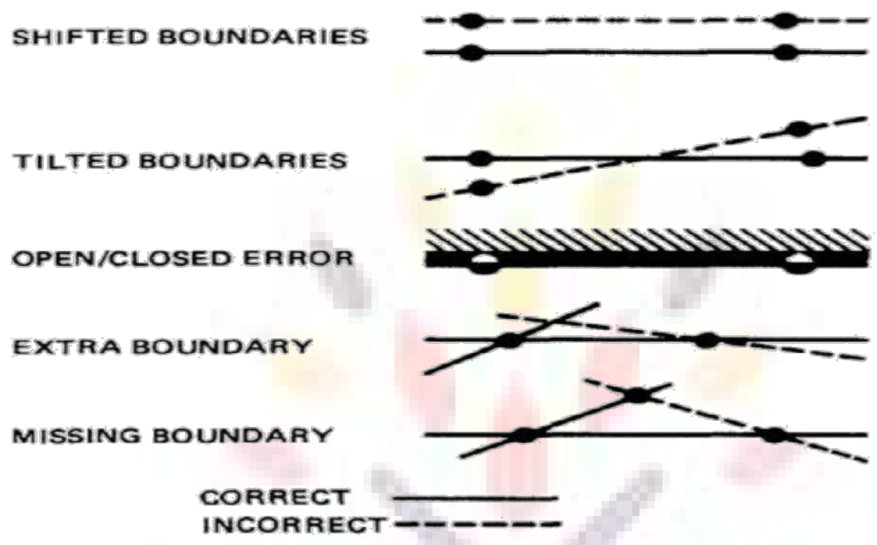


Figure 4.12: Generic Domain Bugs.

TESTING ONE DIMENSIONAL DOMAIN:

The closure can be wrong (i.e., assigned to the wrong domain) or the boundary (a point in this case) can be shifted one way or the other, we can be missing a boundary, or we can have an extra boundary.

Figure 4.13 shows possible domain bugs for a one-dimensional open domain boundary.

In Figure 4.13a we assumed that the boundary was to be open for A. The bug we're looking for is a closure error, which converts > to >= or < to <= (Figure 4.13b). One test (marked x) on the boundary point detects this bug because processing for that point will go to domain A rather than B.

In Figure 4.13c we've suffered a boundary shift to the left. The test point we used for closure detects this bug because the bug forces the point from the B domain, where it should be, to A processing. Note that we can't distinguish between a shift and a closure error, but we do know that we have a bug.

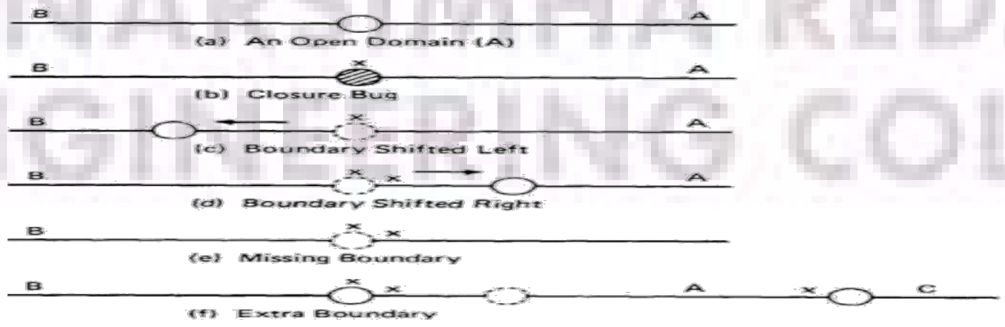


Figure 4.13: One Dimensional Domain Bugs, Open Boundaries.

Figure 4.13d shows a shift the other way. The on point doesn't tell us anything because the boundary shift doesn't change the fact that the test point will be processed in B. To detect this shift we need a point close to the boundary but within A. The boundary is open, therefore by definition, the off point is in A (Open Off Inside).

The same open off point also suffices to detect a missing boundary because what should have been processed in A is now processed in B.
To detect an extra boundary we have to look at two domain boundaries. In this context an extra boundary means that A has been split in two. The two off points that we selected before (one for each boundary) does the job. If point C had been a closed boundary, the on test point at C would do it.

For closed domains look at Figure 4.14. As for the open boundary, a test point on the boundary detects the closure bug. The rest of the cases are similar to the open boundary, except now the strategy requires off points just outside the domain.

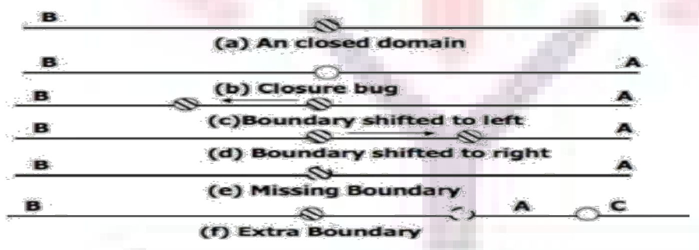


Figure 4.14: One Dimensional Domain Bugs, Closed Boundaries.

TESTING TWO DIMENSIONAL DOMAINS:

Figure 4.15 shows possible domain boundary bugs for a two-dimensional domain.

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.

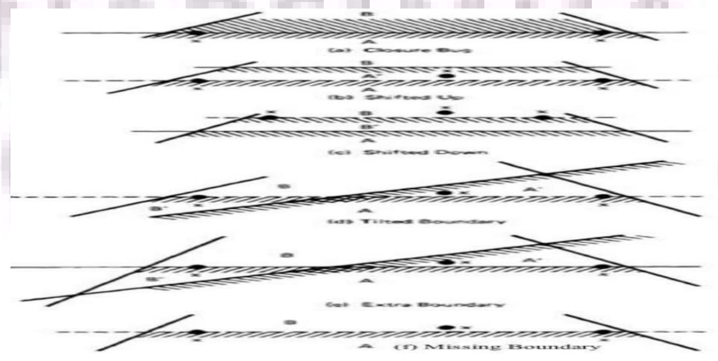


Figure 4.15: Two Dimensional Domain Bugs.

For Closed Boundaries:

Closure Bug: Figure 4.15a shows a faulty closure, such as might be caused by using a wrong operator (for example, x >= k when x > k was intended, or vice versa). The two on points detect this bug because those values will get B rather than A processing.

Shifted Boundary: In Figure 4.15b the bug is a shift up, which converts part of domain B into A processing, denoted by A'. This result is caused by an incorrect constant in a predicate, such as x + y >= 17 when x + y >= 7 was intended. The off point (closed off outside) catches this bug. Figure 4.15c shows a shift down that is caught by the two on points.

Tilted Boundary: A tilted boundary occurs when coefficients in the boundary inequality are wrong. For example, 3x + 7y > 17 when 7x + 3y >

17 was intended. Figure 4.15d has a tilted boundary, which creates erroneous domain segments A' and B'. In this example the bug is caught by the left on point.

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. The extra boundary in Figure 4.15e is caught by two on points, and depending on which way the extra boundary goes, possibly by the off point also.

Missing Boundary: A missing boundary is created by leaving a boundary predicate out. A missing boundary will merge different domains and will cause many test failures although there is only one bug. A missing boundary, shown in Figure 4.15f, is caught by the two on points because the processing for A and B is the same - either A or B processing.

PROCEDURE FOR TESTING: The procedure is conceptually is straight forward. It can be done by hand for two dimensions and for a few domains and practically impossible for more than two variables.

1 Identify input variables.

2 Identify variable which appear in domain defining predicates, such as control flow predicates.

3 Interpret all domain predicates in terms of input variables.

4 For p binary predicates, there are at most 2^p combinations of TRUE-FALSE values and therefore, at most 2^p domains. Find the set of all non null domains. The result is a boolean expression in the predicates consisting a set of AND terms joined by OR's. For example ABC+DEF+GHI...... Where the capital letters denote predicates. Each product term is a set of linear inequality that defines a domain or a part of a multiply connected domains.

5 Solve these inequalities to find all the extreme points of each domain using any of the linear programming methods.

DOMAIN AND INTERFACE TESTING

INTRODUCTION:

Recall that we defined integration testing as testing the correctness of the interface between two otherwise correct components.

Components A and B have been demonstrated to satisfy their component tests, and as part of the act of integrating them we want to investigate possible inconsistencies across their interface.

Interface between any two components is considered as a subroutine call.
We're looking for bugs in that "call" when we do interface testing.
Let's assume that the call sequence is correct and that there are no type incompatibilities.
For a single variable, the domain span is the set of numbers between (and including) the smallest value and the largest value. For every input variable we want (at least): compatible domain spans and compatible closures (Compatible but need not be Equal).

DOMAINS AND RANGE:

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

For most testing, our aim has been to specify input values and to predict and/or confirm output values that result from those inputs.

Interface testing requires that we select the output values of the calling routine *i.e.*caller's range must be compatible with the called routine's domain.

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

caller domain --> caller range	(caller unit test)
caller range --> called domain	(integration test)
called domain --> called range	(called unit test)

CLOSURE COMPATIBILITY:

Assume that the caller's range and the called domain spans the same numbers - for example, 0 to 17.

Figure 4.16 shows the four ways in which the caller's range closure and the called's domain closure can agree.

The thick line means closed and the thin line means open. Figure 4.16 shows the four cases consisting of domains that are closed both on top (17) and bottom (0), open top and closed bottom, closed top and open bottom, and open top and bottom.

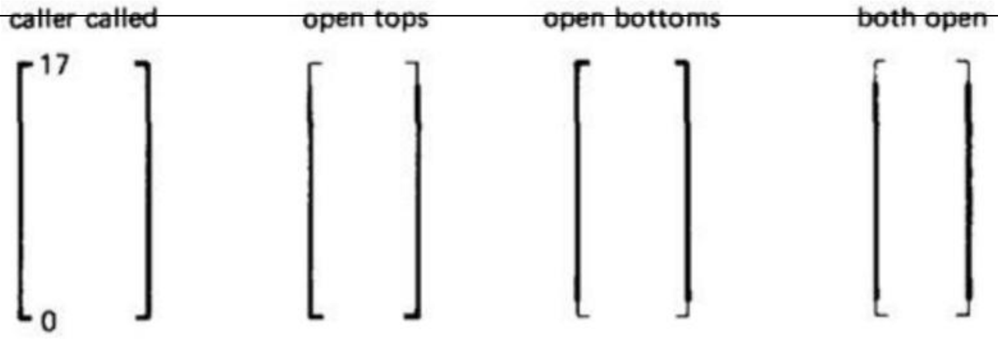


Figure 4.16: Range / Domain Closure Compatibility.

Figure 4.17 shows the twelve different ways the caller and the called can disagree about closure. Not all of them are necessarily bugs. The four cases in which a caller boundary is open and the called is closed (marked with a "?") are probably not buggy. It means that the caller will not supply such values but the called can accept them.

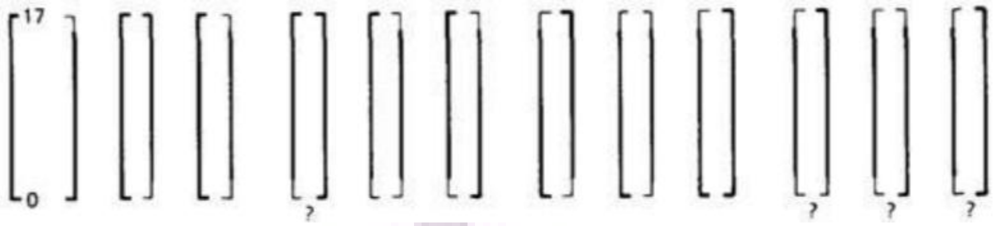


Figure 4.17: Equal-Span Range / Domain Compatibility Bugs.

SPAN COMPATIBILITY:

Figure 4.18 shows three possibly harmless span incompatibilities.

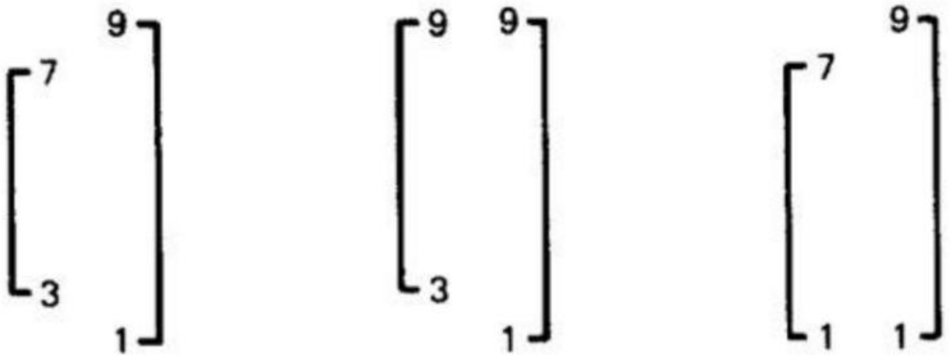


Figure 4.18: Harmless Range / Domain Span incompatibility bug (Caller Span is smaller than Called).

In all cases, the caller's range is a subset of the called's domain. That's not necessarily a bug.

The routine is used by many callers; some require values inside a range and some don't. This kind of span incompatibility is a bug only if the caller expects the called routine to validate the called number for the caller.

Figure 4.19a 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.

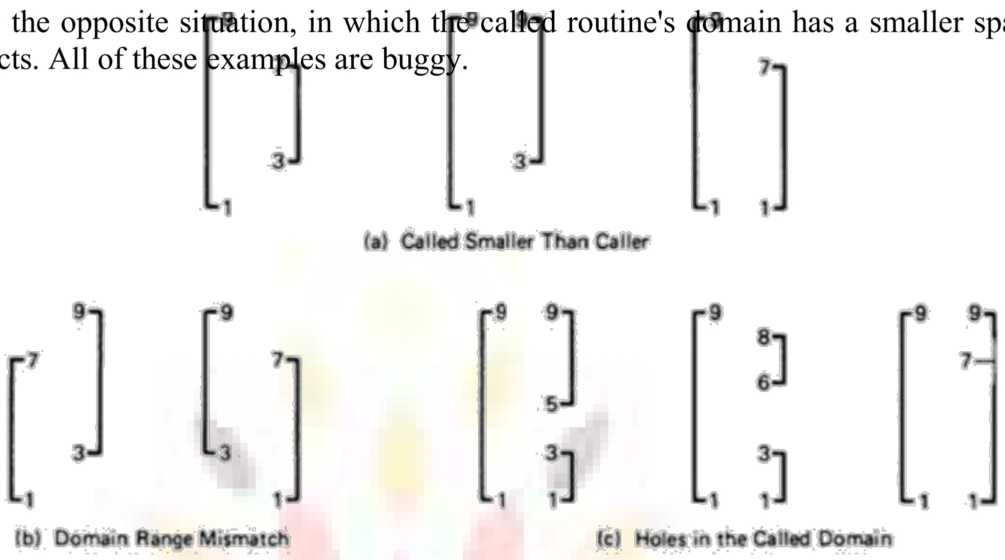

(a) Called Smaller Than Caller
(b) Domain Range Mismatch
(c) Holes in the Called Domain

Figure 4.19: Buggy Range / Domain Mismatches

In Figure 4.19b 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.

Figure 4.19c combines these notions to show various ways we can have holes in the domain: these are all probably buggy.

INTERFACE RANGE / DOMAIN COMPATIBILITY TESTING:

For interface testing, bugs are more likely to concern single variables rather than peculiar combinations of two or more variables.

Test every input variable independently of other input variables to confirm compatibility of the caller's range and the called routine's domain span and closure of every domain defined for that variable.

There are two boundaries to test and it's a one-dimensional domain; therefore, it requires one on and one off point per boundary or a total of two on points and two off points for the domain - pick the off points appropriate to the closure (COOOOI).

Start with the called routine's domains and generate test points in accordance to the domain-testing strategy used for that routine in component testing.

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

UNIT III

PATHS, PATH PRODUCTS AND REGULAR EXPRESSIONS

Paths,Path products and Regular expressions:- path products & path expression, reduction procedure, applications, regular expressions & flow anomaly detection.Logic Based Testing:- overview,decision tables,pathexpressions,kv charts, specifications.

PATH PRODUCTS AND PATH EXPRESSION:

MOTIVATION:

Flow graphs are being an abstract representation of programs.
Any question about a program can be cast into an equivalent question about an appropriate flow graph.Most software development, testing and debugging tools use flow graphs analysis techniques.

PATH PRODUCTS:

Normally flow graphs used to denote only control flow connectivity.
The simplest weight we can give to a link is a name.
Using link names as weights, we then convert the graphical flow graph into an equivalent algebraic like expressions which denotes the set of all possible paths from entry to exit for the flow graph.

Every link of a graph can be given a name.
The link name will be denoted by lower case italic letters In tracing a path or path segment through a flow graph, you traverse a succession of link names.

The name of the path or path segment that corresponds to those links is expressed naturally by concatenating those link names. For example, if you traverse links a,b,c and d along some path, the name for that path segment is abcd. This path name is also called a **path product.** Figure 5.1 shows some examples:

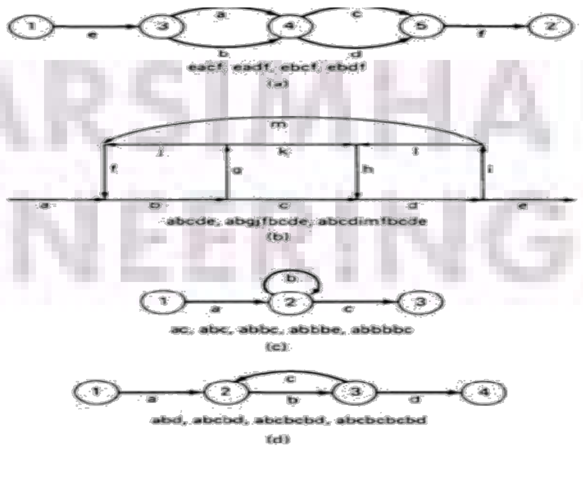

eacf, eadf, ebcf, ebdf
(a)
abcde, abgjfbcde, abcdimfbcde
(b)
ac, abc, abbc, abbbc, abbbbc
(c)
abd, abcbd, abcbcbd, abcbcbcbd
(d)

Figure 5.1: Examples of paths

PATH EXPRESSION:

Consider a pair of nodes in a graph and the set of paths between those node. o Denote that set of paths by Upper case letter such as X,Y. From Figure 5.1c,the members of the path set can be listed as follows:

ac, abc, abbc, abbbc, abbbbc.............

Alternatively, the same set of paths can be denoted by :

ac+abc+abbc+abbbc+abbbbc+...........

The + sign is understood to mean "or" between the two nodes of interest, paths ac, or abc, or abbc, and so on can be taken.

Any expression that consists of path names and "OR"s and which denotes a set of paths between two nodes is called a "**Path Expression".**

PATH PRODUCTS:

The name of a path that consists of two successive path segments is conveniently expressed by the concatenation or **Path Product** of the segment names.

For example, if X and Y are defined as X=abcde,Y=fghij,then the path corresponding to X followed by Y is denoted by

XY=abcdefghij

Similarly,

YX=fghijabcde

aX=aabcde

Xa=abcdea

XaX=abcdeaabcde

If X and Y represent sets of paths or path expressions, their product represents the set of paths that can be obtained by following every element of X by any element of Y in all possible ways. For example,

X = abc + def + ghi

Y = uvw + z

Then,

XY = abcuvw + defuvw + ghiuvw + abcz + defz + ghiz

If a link or segment name is repeated, that fact is denoted by an exponent. The exponent's value denotes the number of repetitions:

$a^1 = a$; $a^2 = aa$; $a^3 = aaa$; $a^n = aaaa \ldots$ n times.

Similarly, if X = abcde then

$X^1 = abcde$
$X^2 = abcdeabcde = (abcde)^2$
$X^3 = abcdeabcdeabcde = (abcde)^2abcde$
$= abcde(abcde)^2 = (abcde)^3$

The path product is not commutative (that is XY!=YX). o The path product is Associative.

RULE 1: A(BC)=(AB)C=ABC

where A,B,C are path names, set of path names or path expressions.

The zeroth power of a link name, path product, or path expression is also needed for completeness. It is denoted by the numeral "1" and denotes the "path" whose length is zero - that is, the path that doesn't have any links.

$a^0 = 1$

$X^0=1$

PATH SUMS:

The "+" sign was used to denote the fact that path names were part of the same set of paths.
The "PATH SUM" denotes paths in parallel between nodes.

Links a and b in Figure 5.1a are parallel paths and are denoted by a + b. Similarly, links c and d are parallel paths between the next two nodes and are denoted by c + d.

The set of all paths between nodes 1 and 2 can be thought of as a set of parallel paths and denoted by eacf+eadf+ebcf+ebdf.

If X and Y are sets of paths that lie between the same pair of nodes, then X+Y denotes the UNION of those set of paths. For example, in Figure 5.2:

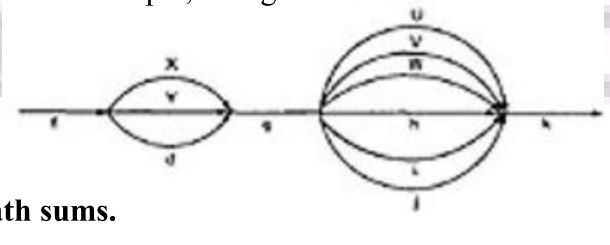


Figure 5.2: Examples of path sums.

The first set of parallel paths is denoted by X + Y + d and the second set by U + V

W + h + i + j. The set of all paths in this flowgraph is f(X + Y + d)g(U + V + W h + i + j)k

The path is a set union operation, it is clearly Commutative and Associative.

- RULE 2: X+Y=Y+X
- RULE 3: (X+Y)+Z=X+(Y+Z)=X+Y+Z

DISTRIBUTIVE LAWS:

The product and sum operations are distributive, and the ordinary rules of multiplication apply; that is

RULE 4: A(B+C)=AB+AC and (B+C)D=BD+CD

Applying these rules to the below Figure 5.1a yields
e(a+b)(c+d)f=e(ac+ad+bc+bd)f = eacf+eadf+ebcf+ebdf

ABSORPTION RULE:
If X and Y denote the same set of paths, then the union of these sets is unchanged; consequently,

RULE 5: X+X=X (Absorption Rule)

If a set consists of paths names and a member of that set is added to it, the "new" name, which is already in that set of names, contributes nothing and can be ignored.

For example,

if X=a+aa+abc+abcd+def then

X+a = X+aa = X+abc = X+abcd = X+def = X

It follows that any arbitrary sum of identical path expressions reduces to the same path expression.

LOOPS:
Loops can be understood as an infinite set of parallel paths. Say that the loop consists of a single link b.then the set of all paths through that loop point is b0+b1+b2+b3+b4+b5+.............

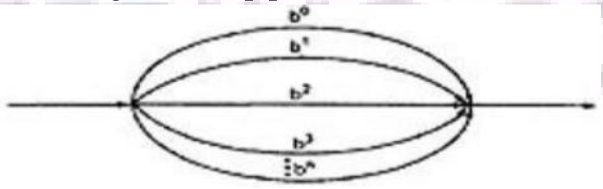


Figure 5.3: Examples of path loops.

This potentially infinite sum is denoted by b* for an individual link and by X*

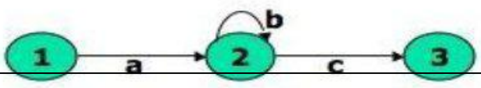


Figure 5.4: Another example of path loops.

The path expression for the above figure is denoted by the

notation: ab*c=ac+abc+abbc+abbbc+................

Evidently,

aa*=a*a=a+ and XX*=X*X=X+

It is more convenient to denote the fact that a loop cannot be taken more than a certain, say n, number of times.

A bar is used under the exponent to denote the fact as follows: $X^{n} =$
$X^{0}+X^{1}+X^{2}+X^{3}+X^{4}+X^{5}+$..................$+X^{n}$

RULES 6 - 16:

o The following rules can be derived from the previous rules:

RULE 6: $X^{n} + X^{\underline{m}} = X^{n}$ if n>m
RULE 6: $X^{n} + X^{\underline{m}} = X^{\underline{m}}$ if m>n
RULE 7: $X^{n}X^{m} = X^{\underline{n+m}}$
RULE 8: $X^{n}X^{*} = X^{*}X^{n} = X^{*}$ RULE 9: $X^{n}X^{+} = X^{+}X^{n} = X^{+}$ RULE
10: $X^{*}X^{+} = X^{+}X^{*} = X^{+}$ RULE 11: $1 + 1 = 1$
RULE 12: $1X = X1 = X$

Following or preceding a set of paths by a path of zero length does not change the set.
RULE 13: $1^{n} = 1^{n} = 1^{*} = 1^{+} = 1$

No matter how often you traverse a path of zero length,It is a path of zero length. RULE 14: $1^{+}+1 = 1^{*}=1$

The null set of paths is denoted by the numeral 0. it obeys the following rules:

RULE 15: X+0=0+X=X

RULE 16: 0X=X0=0

If you block the paths of a graph for or aft by a graph that has no paths , there won’t be any paths.

REDUCTION PROCEDURE:

REDUCTION PROCEDURE ALGORITHM:

This section presents a reduction procedure for converting a flowgraph whose links are labeled with names into a path expression that denotes the set of all entry/exit paths in that flowgraph. The procedure is a node-by-node removal algorithm.
The steps in Reduction Algorithm are as follows:

Combine all serial links by multiplying their path expressions.

Combine all parallel links by adding their path expressions.

Remove all self-loops (from any node to itself) by replacing them with a link of the form X*, where X is the path expression of the link in that loop.

STEPS 4 - 8 ARE IN THE ALGORIHTM'S LOOP:

Select any node for removal other than the initial or final node. Replace it with a set of equivalent links whose path expressions correspond to all the ways you can form a product of the set of in links with the set of out links of that node.

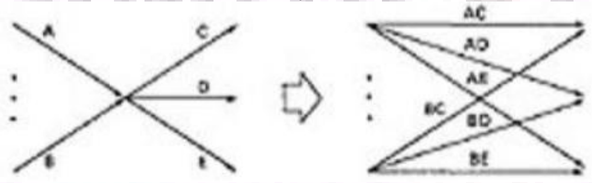
Combine any remaining serial links by multiplying their path expressions.
Combine all parallel links by adding their path expressions.
Remove all self-loops as in step 3.

Does the graph consist of a single link between the entry node and the exit node? If yes, then the path expression for that link is a path expression for the original flowgraph; otherwise, return to step 4.
A flowgraph can have many equivalent path expressions between a given pair of nodes; that is, there are many different ways to generate the set of all paths between two nodes without affecting the content of that set.

The appearance of the path expression depends, in general, on the order in which nodes are removed.

CROSS-TERM STEP (STEP 4):

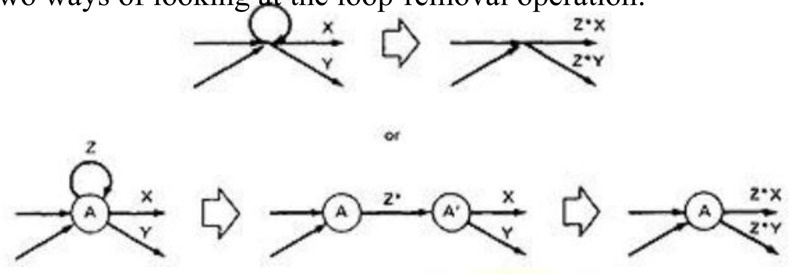
The cross - term step is the fundamental step of the reduction algorithm.
It removes a node, thereby reducing the number of nodes by one.
Successive applications of this step eventually get you down to one entry and one exit node. The following diagram shows the situation at an arbitrary node that has been selected for removal:



From the above diagram, one can infer:
o (a + b)(c + d + e) = ac + ad + + ae + bc + bd + be

LOOP REMOVAL OPERATIONS:

There are two ways of looking at the loop-removal operation:



In the first way, we remove the self-loop and then multiply all outgoing links by Z*.

In the second way, we split the node into two equivalent nodes, call them A and A' and put in a link between them whose path expression is Z*. Then we remove node A' using steps 4 and 5 to yield outgoing links whose path expressions are Z*X and Z*Y.

A REDUCTION PROCEDURE - EXAMPLE:

Let us see by applying this algorithm to the following graph where we remove several nodes in order; that is

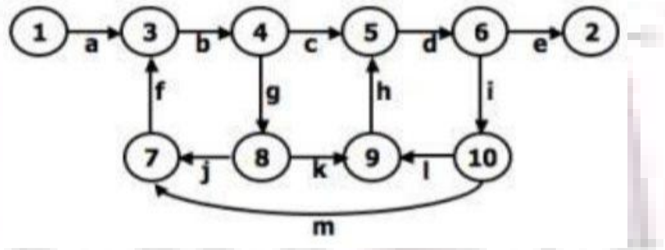
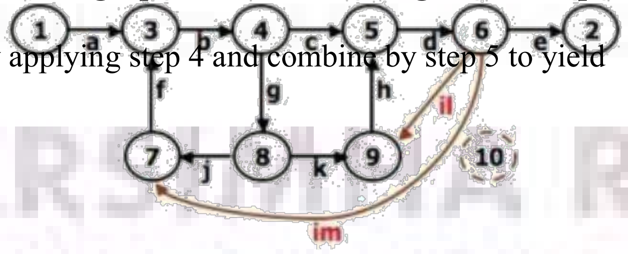
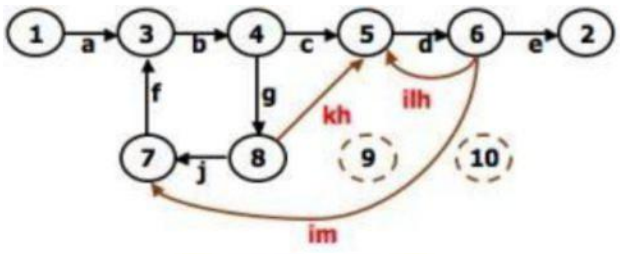


Figure 5.5: Example Flowgraph for demonstrating reduction procedure.

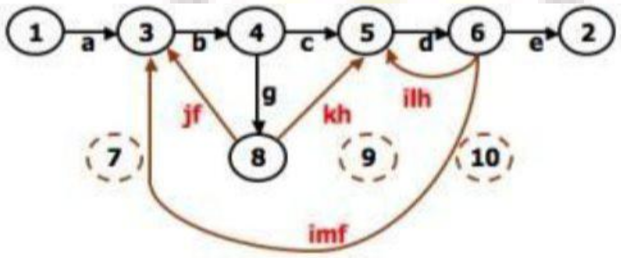
Remove node 10 by applying step 4 and combine by step 5 to yield



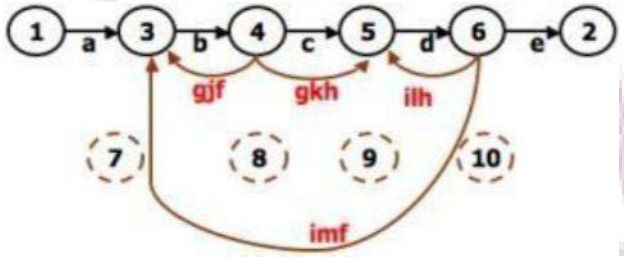
Remove node 9 by applying step4 and 5 to yield



Remove node 7 by steps 4 and 5, as follows:

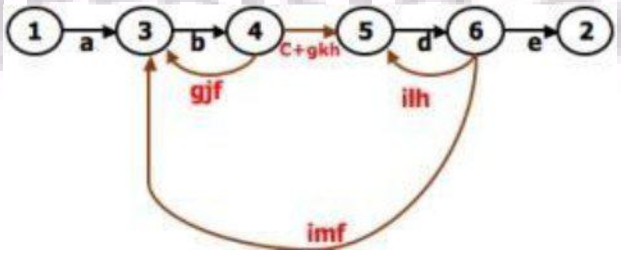


Remove node 8 by steps 4 and 5, to obtain:

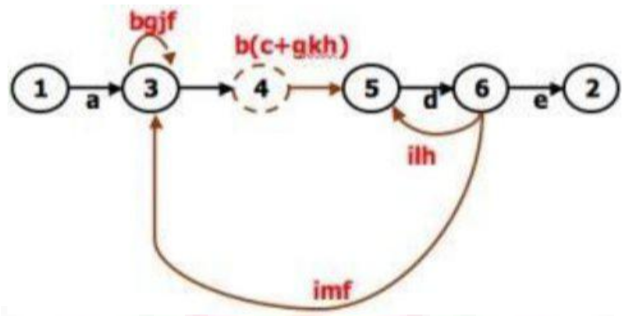


PARALLEL TERM (STEP 6):

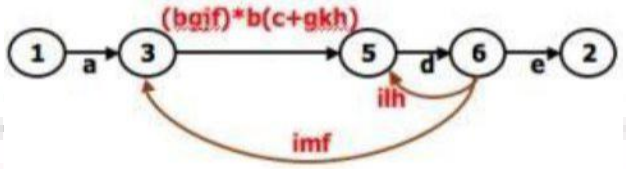
Removal of node 8 above led to a pair of parallel links between nodes 4 and 5. combine them to create a path expression for an equivalent link whose path expression is c+gkh; that is



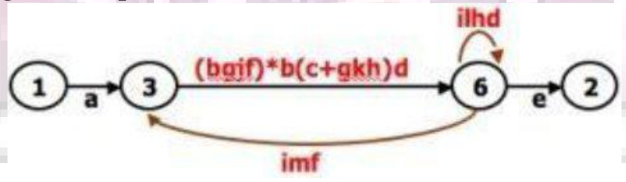
LOOP TERM (STEP 7):
Removing node 4 leads to a loop term. The graph has now been replaced with the following equivalent simpler graph:



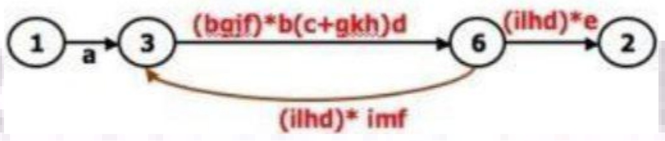
Continue the process by applying the loop-removal step as follows:



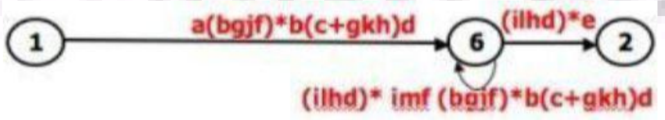
Removing node 5 produces:



Remove the loop at node 6 to yield:



Remove node 3 to yield



Removing the loop and then node 6 result in the following expression: a(bgjf)*b(c+gkh)d((ilhd)*imf(bjgf)*b(c+gkh)d)*(ilhd)*e

You can practice by applying the algorithm on the following flow graphs and generate their respective path expressions:

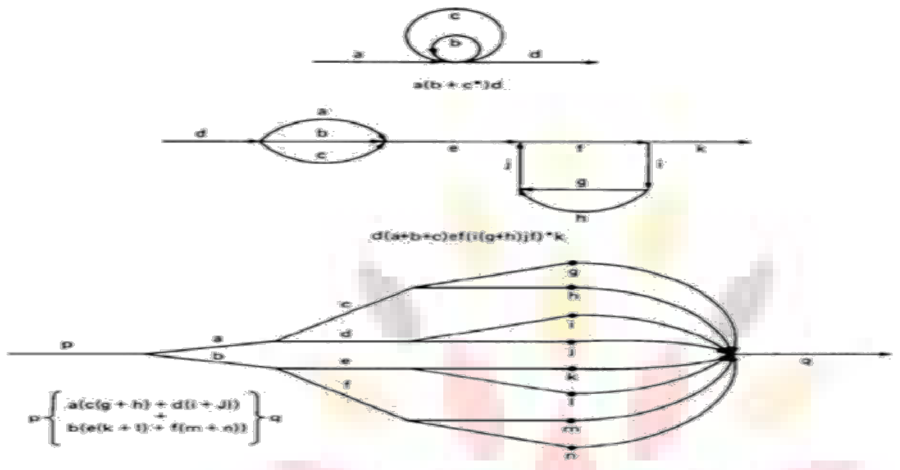


Figure 5.6: Some graphs and their path expressions.

APPLICATIONS:

The purpose of the node removal algorithm is to present one very generalized concept- the path expression and way of getting it.
Every application follows this common pattern:
Convert the program or graph into a path expression.
Identify a property of interest and derive an appropriate set of "arithmetic" rules that characterizes the property.

Replace the link names by the link weights for the property of interest. The path expression has now been converted to an expression in some algebra, such as Ordinary algebra, regular expressions, or boolean algebra. This algebraic expression summarizes the property of interest over the set of all paths. Simplify or evaluate the resulting "algebraic" expression to answer the question you asked.

HOW MANY PATHS IN A FLOW GRAPH ?

The question is not simple. Here are some ways you could ask it:
What is the maximum number of different paths possible?
What is the fewest number of paths possible?
How many different paths are there really?
What is the average number of paths?
Determining the actual number of different paths is an inherently difficult problem because there could be unachievable paths resulting from correlated

and dependent predicates.

If we know both of these numbers (maximum and minimum number of possible paths) we have a good idea of how complete our testing is.

Asking for "the average number of paths" is meaningless.

MAXIMUM PATH COUNT ARITHMETIC:
Label each link with a link weight that corresponds to the number of paths that link represents.

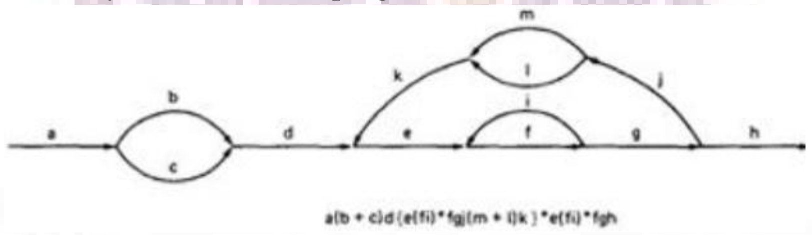
Also mark each loop with the maximum number of times that loop can be taken. If the answer is infinite, you might as well stop the analysis because it is clear that the maximum number of paths will be infinite.

There are three cases of interest: parallel links, serial links, and loops.

Case	Path expression	Weight expression
Parallels	A+B	W_A+W_B
Series	AB	W_AW_B
Loop	$A^{\underline{n}}$	$\sum_{j=0}^{n} W_A^j$

This arithmetic is an ordinary algebra. The weight is the number of paths in each set.
EXAMPLE:

The following is a reasonably well-structured program.

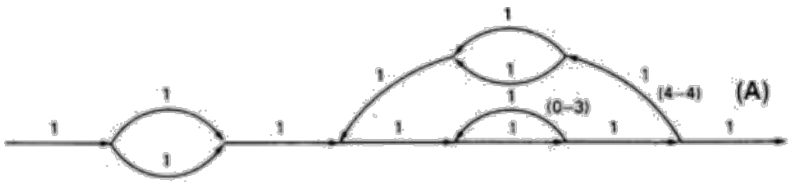


Each link represents a single link and consequently is given a weight of "1" to start. Let’s say the outer loop will be taken exactly four times and inner Loop Can be taken zero or three times Its path expression, with a little work, is:

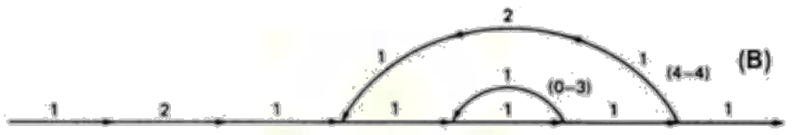
Path expression: a(b+c)d{e(fi)*fgj(m+l)k}*e(fi)*fgh

A: The flow graph should be annotated by replacing the link name with the maximum of paths through that link (1) and also note the number of times for looping.
B: Combine the first pair of parallel loops outside the loop and also the pair in the outer loop.

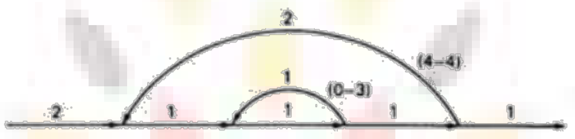
C: Multiply the things out and remove nodes to clear the clutter.



(A)



(B)



(C)

For the Inner Loop:

D:Calculate the total weight of inner loop, which can execute a min. of 0 times and max.

of 3 times. So, it inner loop can be evaluated as follows:

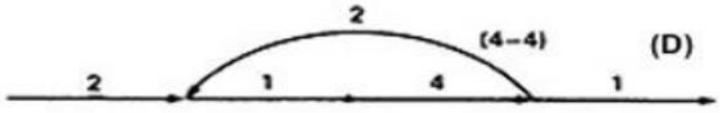
$1^3=1^0+1^1+1^2+1^3=1+1+1+1=4$

E: Multiply the link weights inside the loop: 1 X 4 = 4

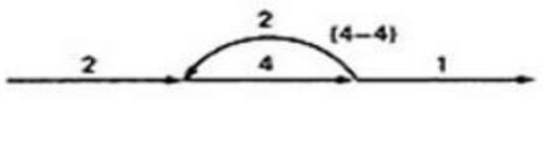
F: Evaluate the loop by multiplying the link wieghts: 2 X 4 = 8.

G: Simpifying the loop further results in the total maximum number of paths in the flowgraph:

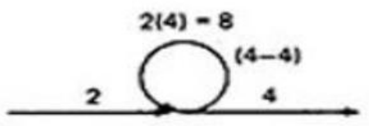
$2 \text{ X } 8^4 \text{ X } 2 = 32{,}768.$



(D)

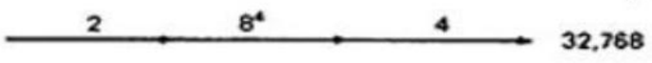


(E)



(F)

(G)



Alternatively, you could have substituted a "1" for each link in the path expression and then simplified, as follows:

a(b+c)d{e(fi)*fgj(m+l)k}*e(fi)*fgh

$1(1 + 1)1(1(1 \times 1)^{3}1 \times 1 \times 1(1 + 1)1)^{4}1(1 \times 1)^{3}1 \times 1 \times 1$
$2(1^{3}1 \times (2))^{4}1^{3}$ -
$2(4 \times 2)^{4} \times 4$
$2 \times 8^{4} \times 4 = 32,768$

This is the same result we got graphically.Actually, the outer loop should be taken exactly four times. That doesn't mean it will be taken zero or four times. Consequently, there is a superfluous "4" on the outlink in the last step. Therefore the maximum number of different paths is 8192 rather than 32,768.

STRUCTURED FLOWGRAPH:
Structured code can be defined in several different ways that do not involve ad-hoc rules such as not using GOTOs.

A structured flowgraph is one that can be reduced to a single link by successive application of the transformations of Figure 5.7.

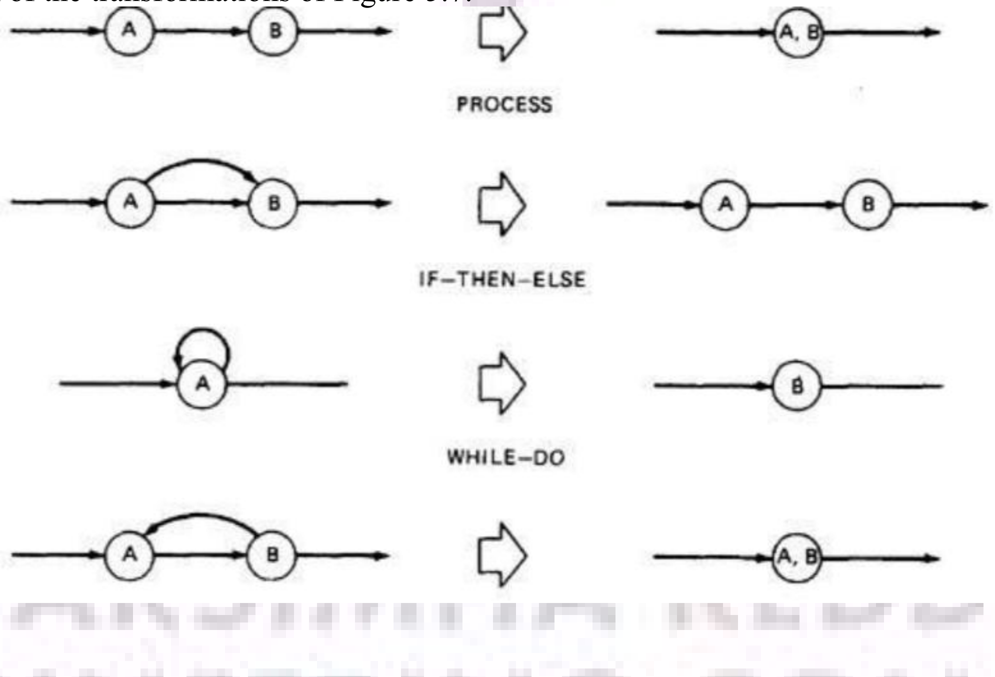


Figure 5.7: Structured Flowgraph Transformations.

The node-by-node reduction procedure can also be used as a test for structured code.Flow graphs that DO NOT contain one or more of the graphs shown below (Figure 5.8) as subgraphs are structured.

Jumping into loops
Jumping out of loops
Branching into decisions
Branching out of decisions

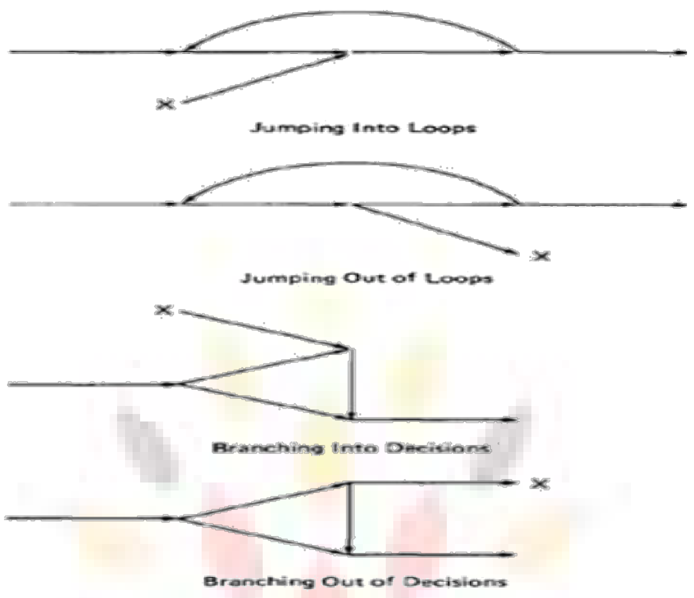
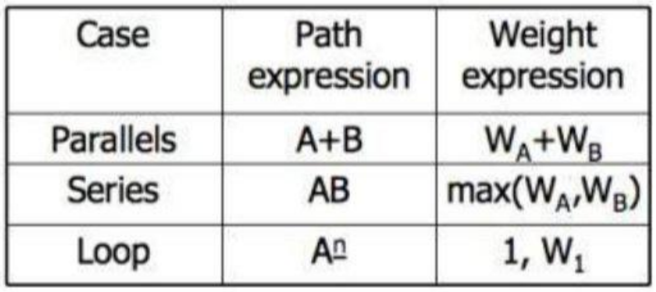


Figure 5.8: Un-structured sub-graphs.

LOWER PATH COUNT ARITHMETIC:
A lower bound on the number of paths in a routine can be approximated for structured flow graphs.

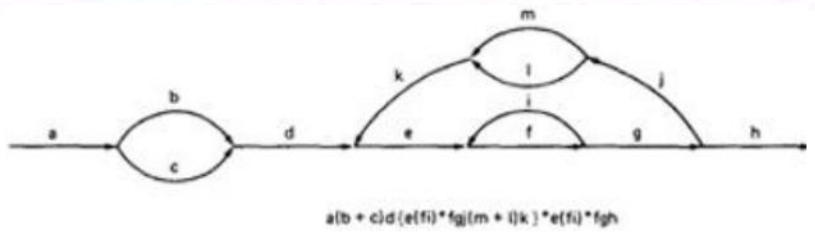
The arithmetic is as follows:

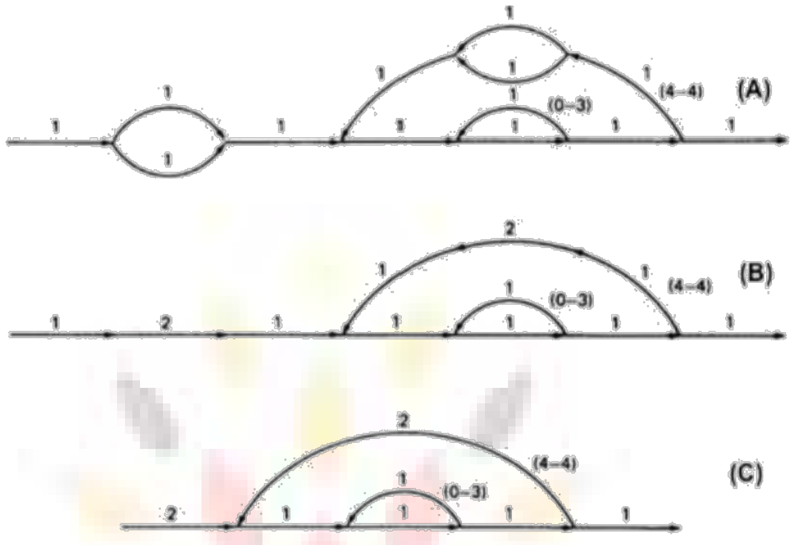


Case	Path expression	Weight expression
Parallels	A+B	W_A+W_B
Series	AB	$\max(W_A,W_B)$
Loop	A^n	$1, W_1$

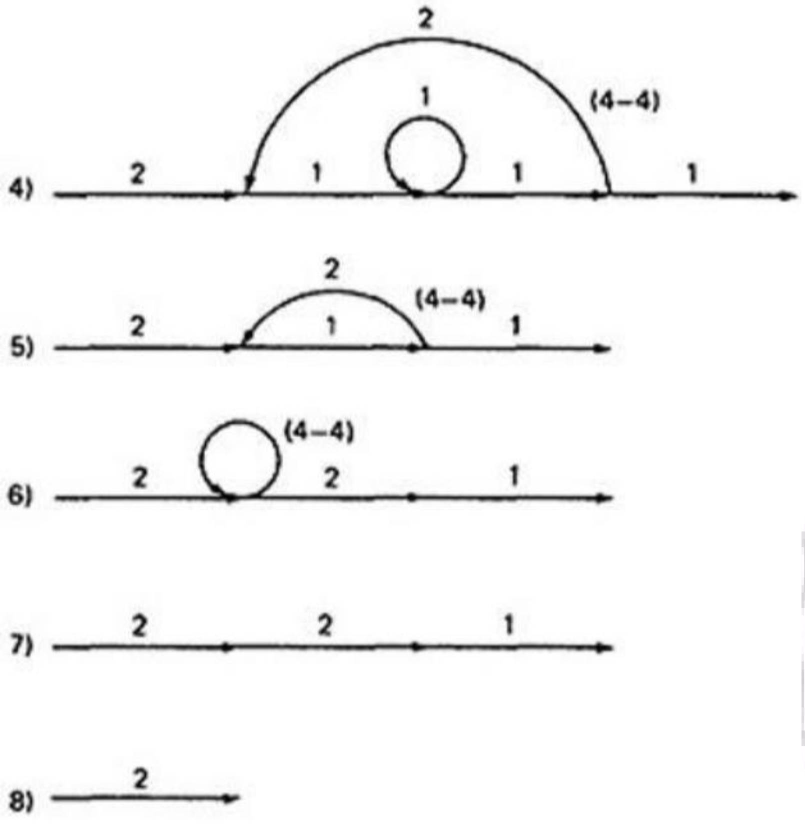
The values of the weights are the number of members in a set of paths.

EXAMPLE:
Applying the arithmetic to the earlier example gives us the identical steps unitl step 3 (C) as below:





From Step 4, the it would be different from the previous example:



If you observe the original graph, it takes at least two paths to cover and that it can be done in two paths.

If you have fewer paths in your test plan than this minimum you probably haven't covered. It's another check.

CALCULATING THE PROBABILITY:

Path selection should be biased toward the low - rather than the high-probability paths.This raises an interesting question:

What is the probability of being at a certain point in a routine?

This question can be answered under suitable assumptions primarily that all probabilities involved are independent, which is to say that all decisions are independent and uncorrelated. We use the same algorithm as before: node-by-node removal of uninteresting nodes.

Weights, Notations and Arithmetic:

Probabilities can come into the act only at decisions (including decisions associated with loops).

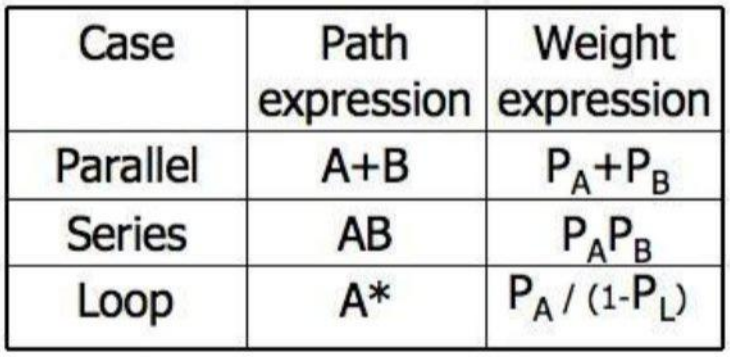
Annotate each outlink with a weight equal to the probability of going in that direction.

Evidently, the sum of the outlink probabilities must equal 1

For a simple loop, if the loop will be taken a mean of N times, the looping probability is N/(N + 1) and the probability of not looping is 1/(N + 1).

A link that is not part of a decision node has a probability of 1.

The arithmetic rules are those of ordinary arithmetic.



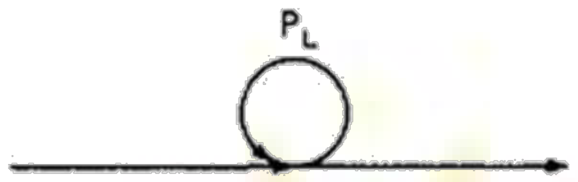
Case	Path expression	Weight expression
Parallel	A+B	P_A+P_B
Series	AB	P_AP_B
Loop	A*	$P_A / (1-P_L)$

In this table, in case of a loop, PA is the probability of the link leaving the loop and PL is the probability of looping.

The rules are those of ordinary probability theory.

If you can do something either from column A with a probability of PA or from column B with a probability PB, then the probability that you do either is PA + PB.

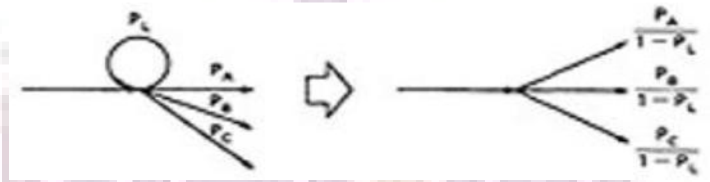
For the series case, if you must do both things, and their probabilities are independent (as assumed), then the probability that you do both is the product of their probabilities.
For example, a loop node has a looping probability of PL and a probability of not looping of PA, which is obviously equal to I - PL.



$$P_A = 1 - P_L$$

$$P_{NEW} = \frac{P_A}{1-P_L} = \frac{1-P_L}{1-P_L} = 1$$

Following the above rule, all we've done is replace the outgoing probability with 1 - so why the complicated rule? After a few steps in which you've removed nodes, combined parallel terms, removed loops and the like, you might find something like this:



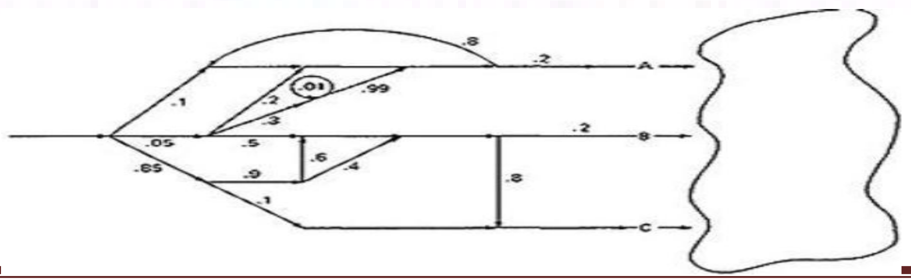
because PL + PA + PB + PC = 1, 1 - PL = PA + PB + PC, and

$$\frac{P_A}{1-P_L} + \frac{P_B}{1-P_L} + \frac{P_C}{1-P_L} = \frac{P_A + P_B + P_C}{1-P_L} = 1$$

which is what we've postulated for any decision. In other words, division by 1 - PL renormalizes the outlink probabilities so that their sum equals unity after the loop is removed.

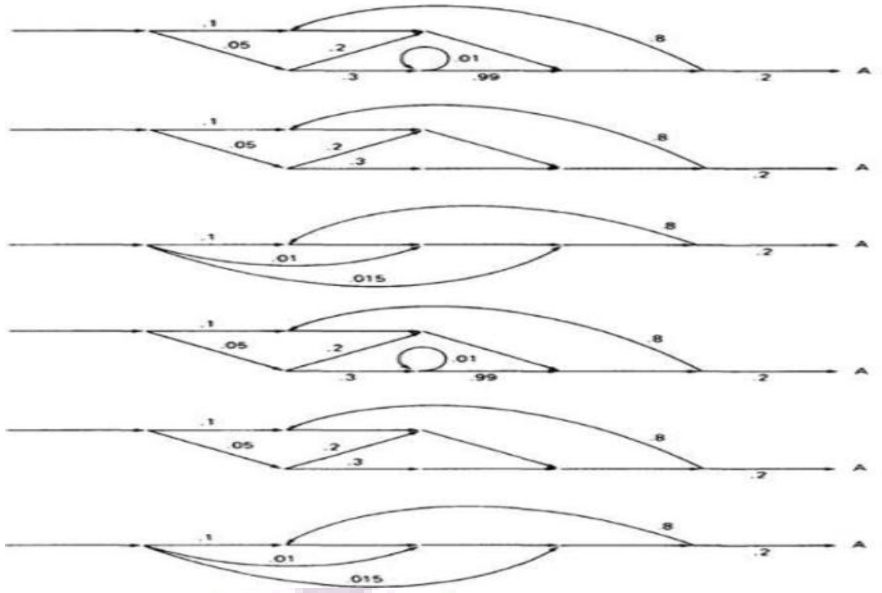
EXAMPLE:

Here is a complicated bit of logic. We want to know the probability associated with cases A, B, and C.

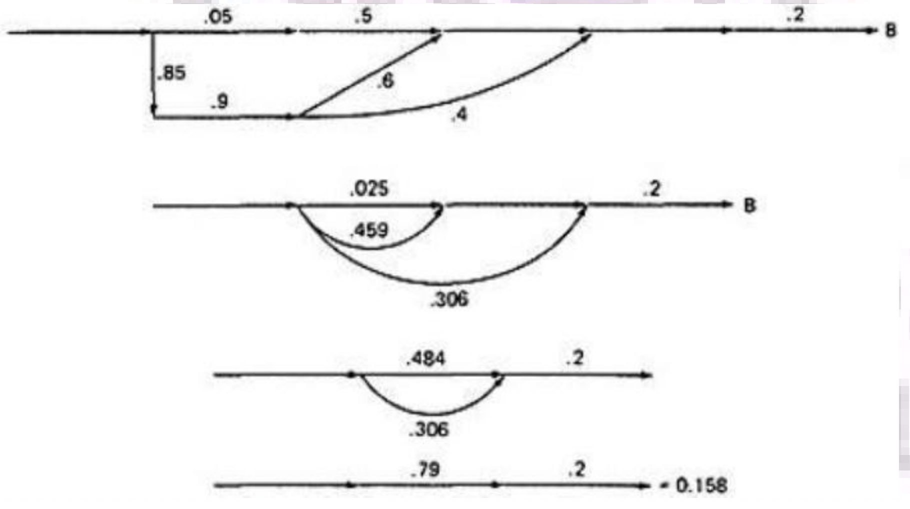


Let us do this in three parts, starting with case A. Note that the sum of the probabilities at each decision node is equal to 1. Start by throwing away anything that isn't on the way to case A, and then apply the reduction procedure. To avoid clutter, we usually leave out probabilities equal to 1.

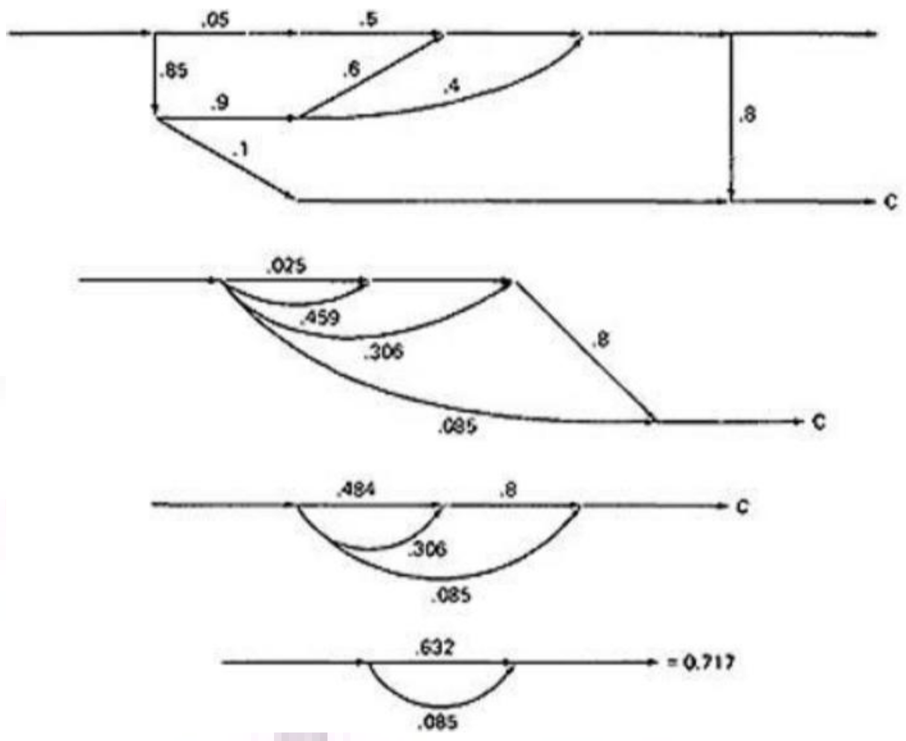
CASE A:



Case B similar:



Case C is similar and should yield a probability of 1 - 0.125 - 0.158 = 0.717:



These checks. It's a good idea when doing this sort of thing to calculate all the probabilities and to verify that the sum of the routine's exit probabilities does equal 1.
If it doesn't, then you've made calculation error or, more likely, you've left out some bra How about path probabilities? That's easy. Just trace the path of interest and multiply the probabilities as you go.

Alternatively, write down the path name and do the indicated arithmetic operation.
Say that a path consisted of links a, b, c, d, e, and the associated probabilities were .2, .5, 1., .01, and I respectively. Path *abcbcbcdeabddea* would have a probability of 5 x 10^{-10}.

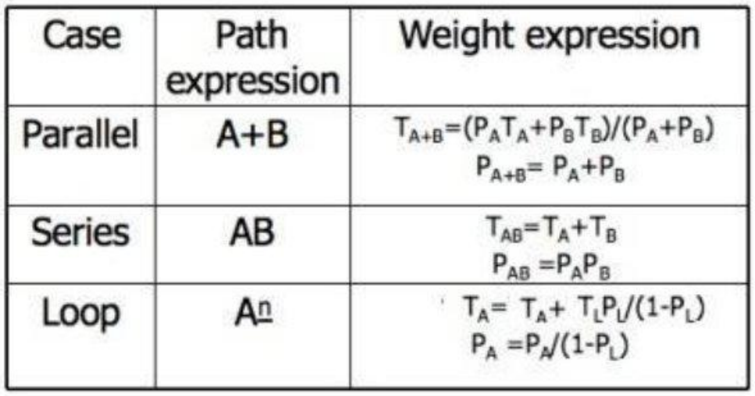
Long paths are usually improbable.

MEAN PROCESSING TIME OF A ROUTINE:

Given the execution time of all statements or instructions for every link in a flowgraph and the probability for each direction for all decisions are to find the mean processing time for the routine as a whole.

The model has two weights associated with every link: the processing time for that link, denoted by **T**, and the probability of that link **P**.

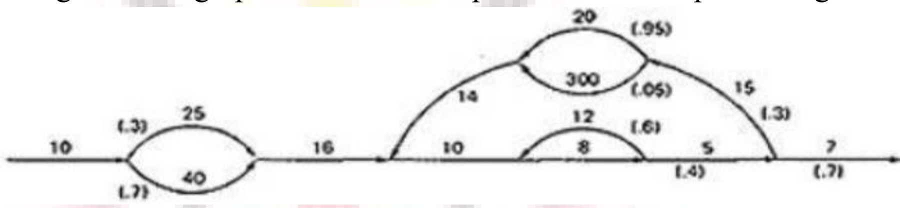
The arithmetic rules for calculating the mean time:



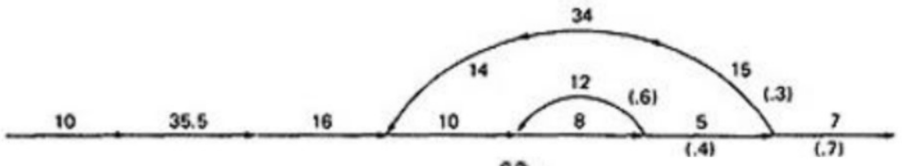
Case	Path expression	Weight expression
Parallel	A+B	$T_{A+B}=(P_AT_A+P_BT_B)/(P_A+P_B)$ $P_{A+B}= P_A+P_B$
Series	AB	$T_{AB}=T_A+T_B$ $P_{AB} =P_AP_B$
Loop	A^n	$T_A= T_A+ T_LP_L/(1-P_L)$ $P_A =P_A/(1-P_L)$

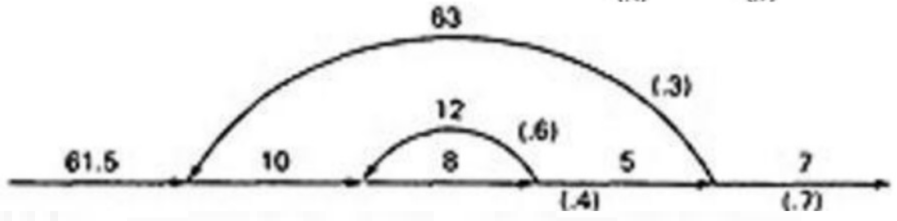
EXAMPLE:

Start with the original flow graph annotated with probabilities and processing time.

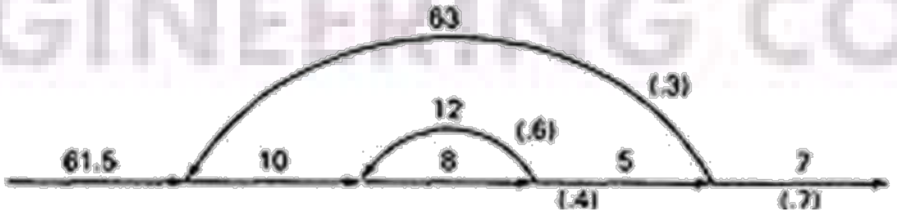


2.Combine the parallel links of the outer loop. The result is just the mean of the processing times for the links because there aren't any other links leaving the first node. Also combine the pair of links at the beginning of the flow graph.



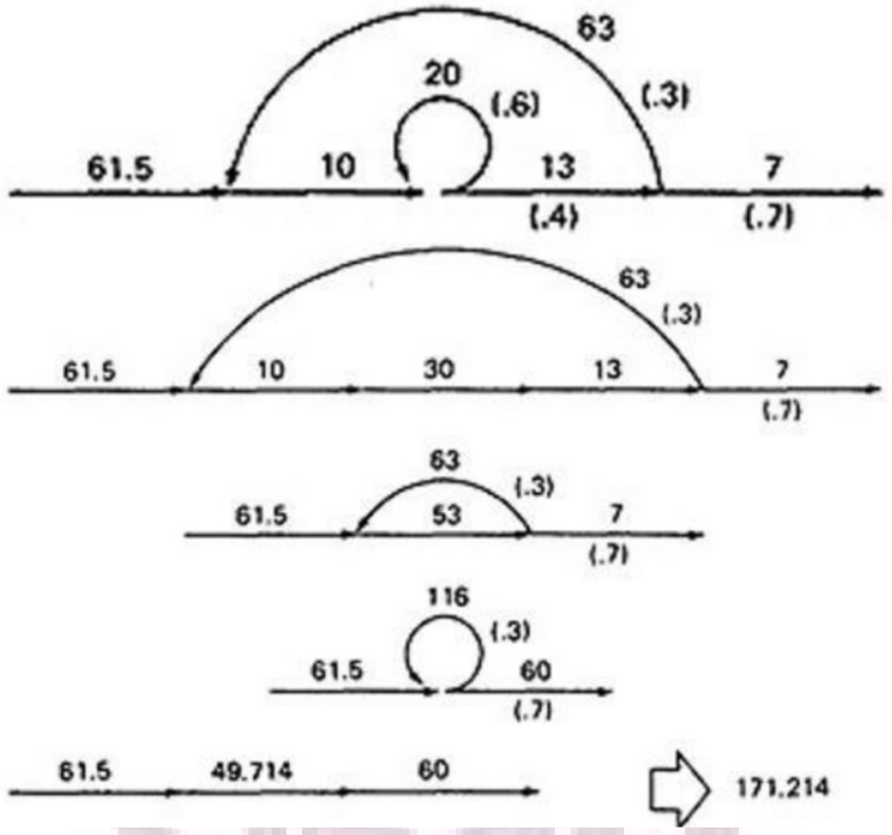


Combine as many serial links as you can.



Use the cross-term step to eliminate a node and to create the inner self - loop.

5.Finally, you can get the mean processing time, by using the arithmetic rules as follows:



PUSH/POP, GET/RETURN:
This model can be used to answer several different questions that can turn up in debugging.
It can also help decide which test cases to design.

The question is:

Given a pair of complementary operations such as PUSH (the stack) and POP (the stack), considering the set of all possible paths through the routine, what is the net effect of the routine? PUSH or POP? How many times? Under what conditions?

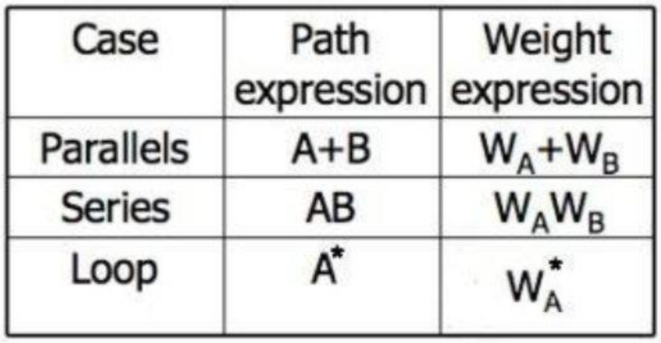
Here are some other examples of complementary operations to which this model applies:
GET/RETURN a resource block.

OPEN/CLOSE a file.

START/STOP a device or process.

EXAMPLE 1 (PUSH / POP):

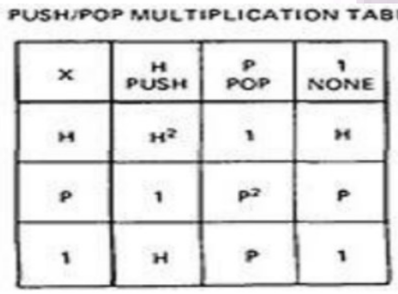
Here is the Push/Pop Arithmetic:



Case	Path expression	Weight expression
Parallels	A+B	W_A+W_B
Series	AB	W_AW_B
Loop	A^*	W_A^*

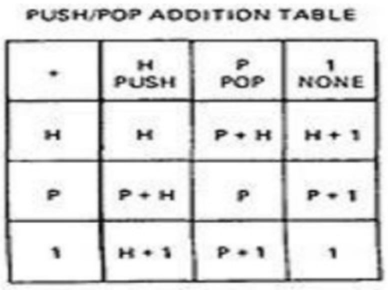
The numeral 1 is used to indicate that nothing of interest (neither PUSH nor POP) occurs on a given link.

"H" denotes PUSH and "P" denotes POP. The operations are commutative, associative, and distributive.



PUSH/POP MULTIPLICATION TABLE

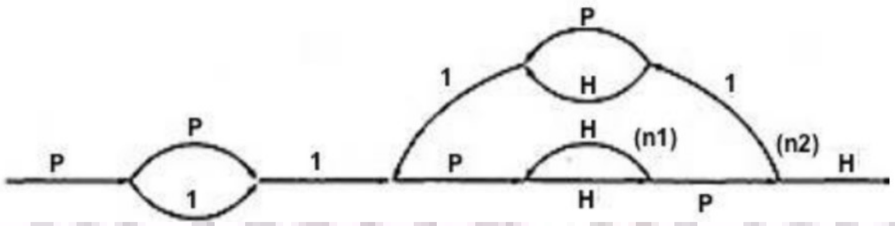
×	H PUSH	P POP	1 NONE
H	H^2	1	H
P	1	P^2	P
1	H	P	1



PUSH/POP ADDITION TABLE

+	H PUSH	P POP	1 NONE
H	H	P + H	H + 1
P	P + H	P	P + 1
1	H + 1	P + 1	1

Consider the following flow graph:



$P(P + 1)1\{P(HH)^{n1}HP1(P + H)1\}^{n2}P(HH)^{n1}HPH$

Simplifying by using the arithmetic tables,

$(P^2 + P)\{P(HH)^{n1}(P + H)\}^{n1}(HH)^{n1}$
$(P^2 + P)\{H^{2n1}(P^2 + 1)\}^{n2}H^{2n1}$

Below Table 5.9 shows several combinations of values for the twolooping terms - M1 is the number of times the inner loop will be taken and M2 the number of times the outer loop will be taken.

M_1	M_2	PUSH/POP
0	0	$P + P^2$
0	1	$P + P^2 + P^3 + P^4$
0	2	$\sum_{1}^{6} P^i$
0	3	$\sum_{1}^{8} P^i$
1	0	$1 + H$
1	1	$\sum_{0}^{3} H^i$
1	2	$\sum_{0}^{5} H^i$
1	3	$\sum_{0}^{7} H^i$
2	0	$H^2 + H^3$
2	1	$\sum_{4}^{7} H^i$
2	2	$\sum_{6}^{11} H^i$
2	3	$\sum_{8}^{15} H^i$

Figure 5.9: Result of the PUSH / POP Graph Analysis.

These expressions state that the stack will be popped only if the inner loop is not taken.

The stack will be left alone only if the inner loop is iterated once, but it may also be pushed.

For all other values of the inner loop, the stack will only be pushed.

EXAMPLE 2 (GET / RETURN):

Exactly the same arithmetic tables used for previous example are used for GET / RETURN a buffer block or resource, or, in fact, for any pair of complementary operations in which the total number of operations in either direction is cumulative.

The arithmetic tables for GET/RETURN are:

Multiplication Table

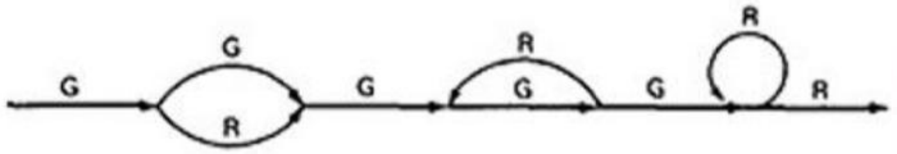
×	G	R	1
G	G^2	1	G
R	1	R^2	R
1	G	R	1

Addition Table

+	G	R	1
G	G	G + R	G + 1
R	G + R	R	R + 1
1	G + 1	R + 1	1

"G" denotes GET and "R" denotes RETURN.

Consider the following flowgraph:



$G(G + R)G(GR)^*GGR^*R$

$G(G + R)G^3R^*R$
$(G + R)G^3R^*$
$(G^4 + G^2)R^*$

This expression specifies the conditions under which the resources will be balanced on leaving the routine.

If the upper branch is taken at the first decision, the second loop must be taken four times.

If the lower branch is taken at the first decision, the second loop must be taken twice.

For any other values, the routine will not balance. Therefore, the first loop does not have to be instrumented to verify this behavior because its impact should be nil.

LIMITATIONS AND SOLUTIONS:

The main limitation to these applications is the problem of unachievable paths.
The node-by-node reduction procedure, and most graph-theory-based algorithms work well when all paths are possible, but may provide misleading results when some paths are unachievable.

The approach to handling unachievable paths (for any application) is to partition the graph into subgraphs so that all paths in each of the subgraphs are achievable.

The resulting subgraphs may overlap, because one path may be common to several different subgraphs.
Each predicate's truth-functional value potentially splits the graph into two subgraphs.
For n predicates, there could be as many as 2^n subgraphs.

REGULAR EXPRESSIONS AND FLOW ANOMALY DETECTION:

THE PROBLEM:
The generic flow-anomaly detection problem (note: not just data-flow anomalies, but any flow anomaly) is that of looking for a specific sequence of options considering all possible paths through a routine.

Let the operations be SET and RESET, denoted by s and r respectively, and we want to know if there is a SET followed immediately a SET or a RESET followed immediately by a RESET (an *ss* or an *rr* sequence).
Some more application examples:
A file can be opened (o), closed (c), read (r), or written (w). If the file is read or written to after it's been closed, the sequence is nonsensical. Therefore, *cr* and *cw* are anomalous. Similarly, if the file is read before it's been written, just after opening, we may have a bug. Therefore, *or* is also anomalous. Furthermore, *oo* and *cc*, though not actual bugs, are a waste of time and therefore should also be examined.
A tape transport can do a rewind (d), fast-forward (f), read (r), write (w), stop (p), and skip (k). There are rules concerning the use of the transport; for example, you cannot go from rewind to fast-forward without an intervening stop or from rewind or fast-forward to read or write without an intervening stop. The following sequences are anomalous: *df*, *dr*, *dw*, *fd*, and *fr*. Does the flowgraph lead to anomalous sequences on any path? If so, what sequences and under what circumstances?

The data-flow anomalies discussed in Unit 4 requires us to detect the *dd*, *dk*, *kk*, and *ku* sequences. Are there paths with anomalous data flows?

THE METHOD:
Annotate each link in the graph with the appropriate operator or the null

operator 1.

Simplify things to the extent possible, using the fact that a + a = a and 12 = 1. o You now have a regular expression that denotes all the possible sequences of operators in that graph. You can now examine that regular expression for the sequences of interest.

o **EXAMPLE:** Let A, B, C, be nonempty sets of character sequences whose smallest string is at least one character long. Let T be a two-character string of characters. Then if T is a substring of (i.e., if T appears within) AB^nC, then T will appear in
AB^2C. (**HUANG's Theorem**)
As an example, let

A = *pp*
B = *srr*

C = *rp* T = *ss*

The theorem states that *ss* will appear in $pp(srr)^n rp$ if it appears in $pp(srr)^2 rp$.
However, let

A = $p + pp + ps$
B = $psr + ps(r + ps)$ C = rp
T=P^4

Is it obvious that there is a p^4 sequence in AB^nC? The theorem states that we have only to look at

$(p + pp + ps)[psr + ps(r + ps)]^2 rp$

Multiplying out the expression and simplifying shows that
there is no p^4 sequence.

Incidentally, the above observation is an informal proof of the wisdom of looping twice discussed in Unit 2. Because data-flow anomalies are represented by two-character sequences, it follows the above theorem that looping twice is what you need to do to find such anomalies.

LIMITATIONS:
Huang's theorem can be easily generalized to cover sequences of greater length than two characters. Beyond three characters, though, things get complex and this method has probably reached its utilitarian limit for manual application.

There are some nice theorems for finding sequences that occur at the beginnings and ends of strings but no nice algorithms for finding strings buried in an expression.

Static flow analysis methods can't determine whether a path is or is not achievable. Unless the flow analysis includes symbolic execution or similar techniques, the impact of unachievable paths will not be included in the analysis.

The flow-anomaly application, for example, doesn't tell us that there will be a flow anomaly - it tells us that if the path is achievable, then there will be a flow anomaly. Such analytical problems go away, of course, if you take the trouble to design routines for which all paths are achievable.

LOGIC BASED TESTING

OVERVIEW OF LOGIC BASED TESTING:

INTRODUCTION:

The functional requirements of many programs can be specified by **decision tables**, which provide a useful basis for program and test design.

Consistency and completeness can be analyzed by using boolean algebra, which can also be used as a basis for test design. Boolean algebra is trivialized by using

Karnaugh-Veitch charts.

"Logic" is one of the most often used words in programmers' vocabularies but one of their least used techniques.

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.

Logic has been, for several decades, the primary tool of hardware logic designers. o Many test methods developed for hardware logic can be adapted to software logic testing. Because hardware testing automation is 10 to 15 years ahead of software testing automation, hardware testing methods and its associated theory is a fertile ground for software testing methods.

As programming and test techniques have improved, the bugs have shifted closer to the process front end, to requirements and their specifications. These bugs range from 8% to 30% of the total and because they're first-in and last-out, they're the costliest of all.

The trouble with specifications is that they're hard to express.
Boolean algebra (also known as the sentential calculus) is the most basic of all logic systems.

o Higher-order logic systems are needed and used for formal specifications.

o Much of logical analysis can be and is embedded in tools. But these tools incorporate methods to simplify, transform, and check specifications, and the methods are to a large extent based on boolean algebra.

KNOWLEDGE BASED SYSTEM:

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.

One implementation of knowledge-based systems is to incorporate the expert's knowledge into a set of rules. The user can then provide data and ask questions based on that data.

The user's data is processed through the rule base to yield conclusions (tentative or definite) and requests for more data. The processing is done by a program called the **inference engine**.
Understanding knowledge-based systems and their validation problems requires an understanding of formal logic.

Decision tables are extensively used in business data processing; Decision-table preprocessors as extensions to COBOL are in common use; boolean algebra is embedded in the implementation of these processors.

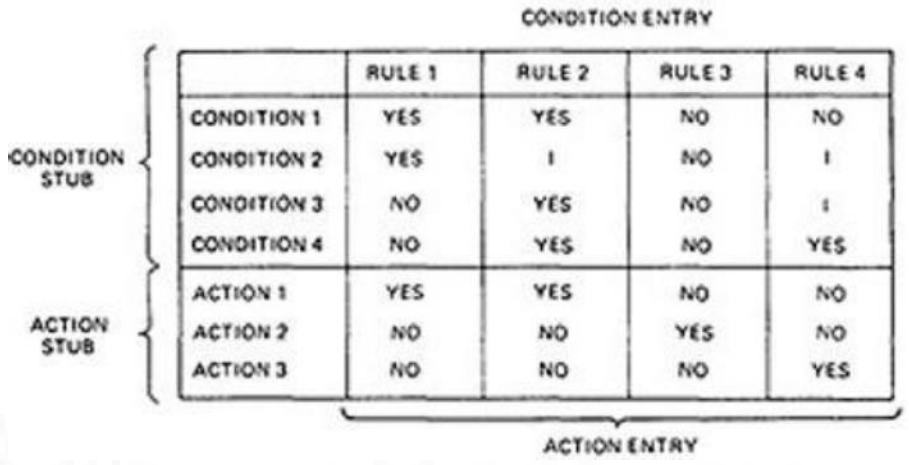
Although programmed tools are nice to have, most of the benefits of boolean algebra can be reaped by wholly manual means if you have the right conceptual tool: the Karnaugh-Veitch diagram is that conceptual tool.

DECISION TABLES:

Figure 6.1 is a limited - entry decision table. It consists of four areas called the condition stub, the condition entry, the action stub, and the action entry.

Each column of the table is a rule that specifies the conditions under which the actions named in the action stub will take place.

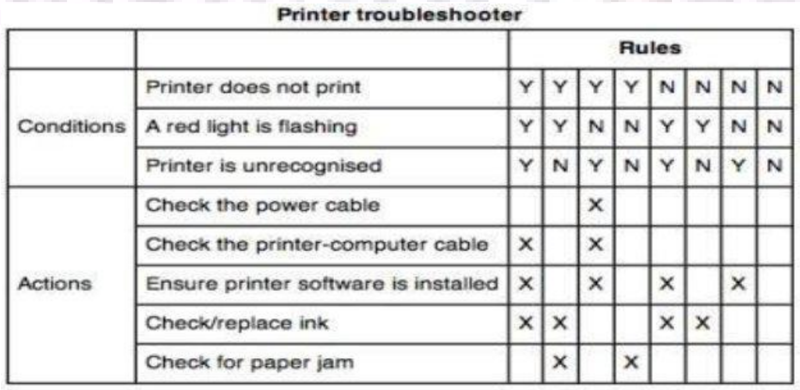
The condition stub is a list of names of conditions.


CONDITION ENTRY
CONDITION STUB
ACTION STUB
ACTION ENTRY

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

Figure 6.1 : Examples of Decision Table.

A more general decision table can be as below:



Printer troubleshooter

		Rules							
Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognised	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

Figure 6.2 : Another Examples of Decision Table.

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.

The table in Figure 6.1 can be translated as follows:

Action 1 will take place if conditions 1 and 2 are met and if conditions 3 and 4 are not met (rule or if conditions 1, 3, and 4 are met (rule 2). "Condition" is another word for predicate.

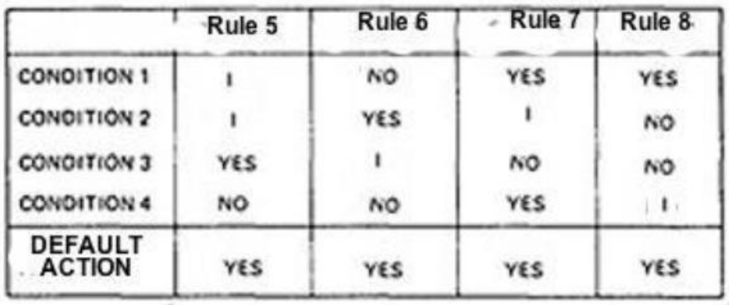
Decision-table uses "condition" and "satisfied" or "met". Let us use "predicate" and

TRUE / FALSE.

Now the above translations become:

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).

In addition to the stated rules, we also need a **Default Rule** that specifies the default action to be taken when all other rules fail. The default rules for Table in Figure 6.1 is shown in Figure 6.3



	Rule 5	Rule 6	Rule 7	Rule 8
CONDITION 1	I	NO	YES	YES
CONDITION 2	I	YES	I	NO
CONDITION 3	YES	I	NO	NO
CONDITION 4	NO	NO	YES	I
DEFAULT ACTION	YES	YES	YES	YES

Figure 6.3 : The default rules of Table in Figure 6.1

DECISION-TABLE PROCESSORS:

Decision tables can be automatically translated into code and, as such, are a higher-order language 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 Decision tables have become a useful tool in the programmers kit, in business data processing.

DECISION-TABLES AS BASIS FOR TEST CASE DESIGN:

The specification is given as a decision table or can be easily converted into one.

The order in which the predicates are evaluated does not affect interpretation of the rules or the resulting action - i.e., an arbitrary permutation of the predicate order will not, or should not, affect which action takes place.
The order in which the rules are evaluated does not affect the resulting action - i.e., an arbitrary permutation of rules will not, or should not, affect which action takes place.

Once a rule is satisfied and an action selected, no other rule need be examined.

If several actions can result from satisfying a rule, the order in which the actions are executed doesn't matter.

DECISION-TABLES AND STRUCTURE:

Decision tables can also be used to examine a program's structure. o Figure 6.4 shows
a program segment that consists of a decision tree.

o These decisions, in various combinations, can lead to actions 1, 2, or 3.

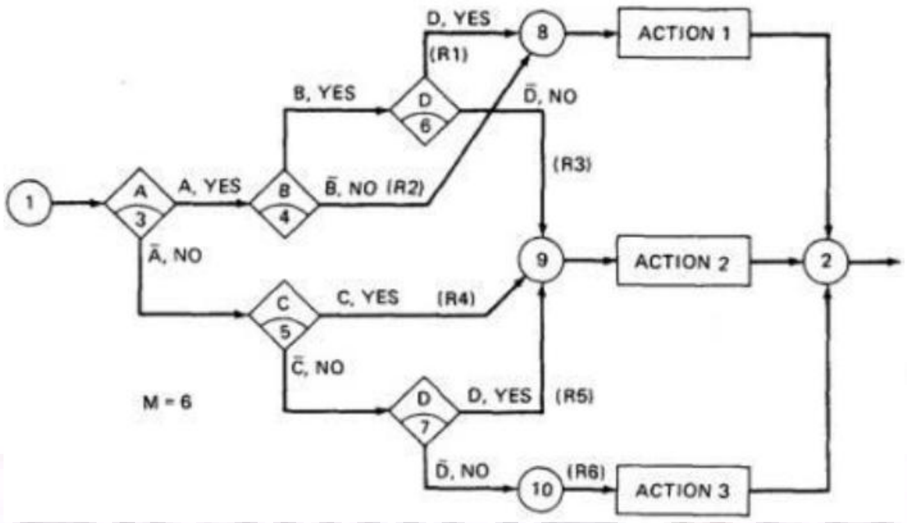
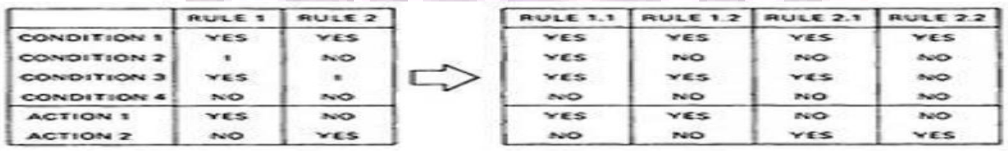


Figure 6.4 : A Sample Program
If the decision appears on a path, put in a YES or NO as appropriate. If the decision does not appear on the path, put in an I, Rule 1 does not contain decision C, therefore its entries are: YES, YES, I, YES.
The corresponding decision table is shown in Table 6.1

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

Table 6.1: Decision Table corresponding to Figure 6.4

As an example, expanding the immaterial cases results as below:



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

RULE 1.1	RULE 1.2	RULE 2.1	RULE 2.2
YES	YES	YES	YES
YES	NO	NO	NO
YES	YES	YES	NO
NO	NO	NO	NO
YES	YES	NO	NO
NO	NO	YES	YES

Table 6.2: Expansion of Table 6.1

Similalrly, If we expand the immaterial cases for the above Table 6.1, it results in Table 6.2 as below:

	R 1	RULE 2	R 3	RULE 4	R 5	R 6
CONDITION A	YY	YYYY	YY	NNNN	NN	NN
CONDITION B	YY	NNNN	YY	YYNN	NY	YN
CONDITION C						
CONDITION	YN	NNYY	YN	YYYY	NN	NN
D	YY	YNNY	NN	NYYN	YY	NN

Sixteen cases are represented in Table 6.1, 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. We can find the bug that way.

ANOTHER EXAMPLE - A TROUBLE SOME PROGRAM:

Consider the following specification whose putative flowgraph is shown in Figure6.5:

If condition A is met, do process A1 no matter what other actions are taken or what other conditions are met.
If condition B is met, do process A2 no matter what other actions are taken or what other conditions are met.
If condition C is met, do process A3 no matter what other actions are taken or what other conditions are met.
If none of the conditions is met, then do processes A1, A2, and A3.

When more than one process is done, process A1 must be done first, then A2, and then A3. The only permissible cases are: (A1), (A2), (A3), (A1,A3), (A2,A3) and (A1,A2,A3).

Figure 6.5 shows a sample program with a bug.

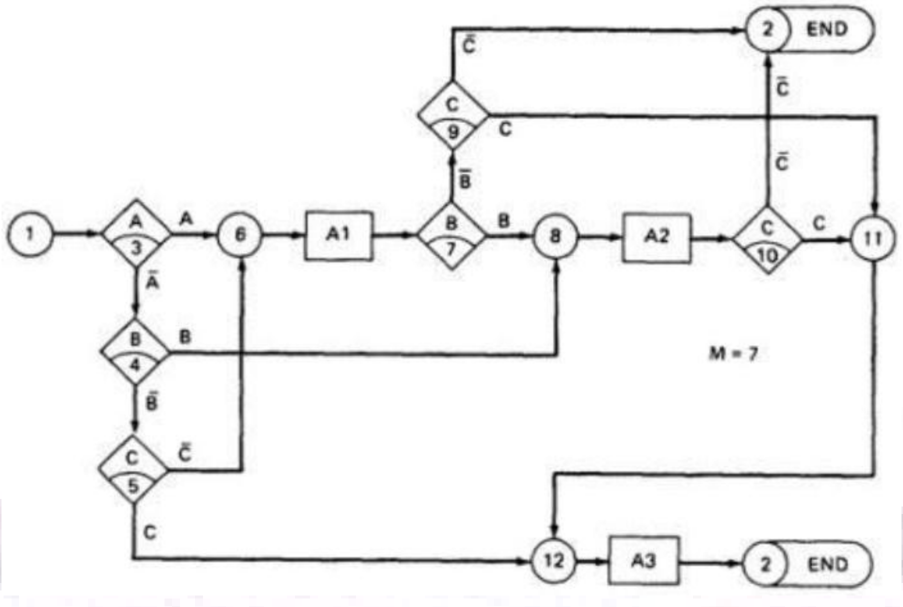
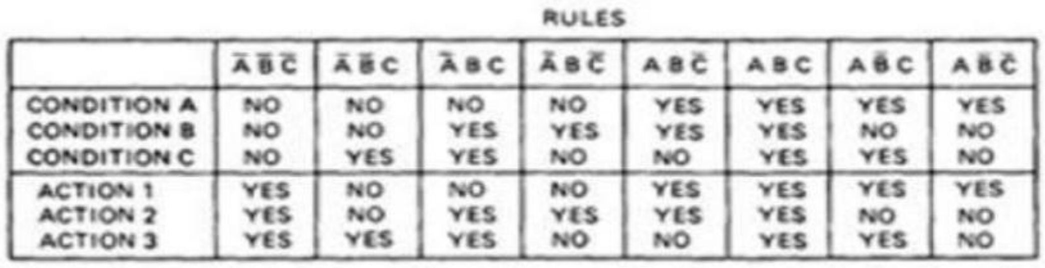


Figure 6.5 : A Troublesome Program

The programmer tried to force all three processes to be executed for the $\bar{A}\bar{B}\bar{C}$cases but forgot that the B and C predicates would be done again, thereby bypassing processes A2 and A3.

Table 6.3 shows the conversion of this flow graph into a decision table after expansion.



RULES	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}C$	$\bar{A}BC$	$\bar{A}B\bar{C}$	$AB\bar{C}$	ABC	$A\bar{B}C$	$A\bar{B}\bar{C}$
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	NO	YES	YES	YES	YES	NO	NO
ACTION 3	YES	YES	YES	NO	NO	YES	YES	NO

Table 6.3: Decision Table for Figure 6.5

PATH EXPRESSIONS:
GENERAL:

Logic-based testing is structural testing when it's applied to structure (e.g., control flow graph of an implementation); it's functional testing when it's applied to a specification.

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

A **predicate** is implemented as a process whose outcome is a truth-functional value.
For our purpose, logic-based testing is restricted to binary predicates.
We start by generating path expressions by path tracing as in Unit V, but this time, our purpose is to convert the path expressions into boolean algebra, using the predicates' truth values (e.g., A and $\bar{A}$) as weights.

BOOLEAN ALGEBRA:
STEPS:

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 (say A) and the NO or FALSE branch with the same letter overscored (say $\bar{A}$).

The truth value of a path is the product of the individual labels. Concatenation or products mean "AND". For example, the straight-through path of Figure 6.5, which goes via nodes 3, 6, 7, 8, 10, 11, 12, and 2, has a truth value of ABC. The path via nodes 3, 6, 7, 9 and 2 has a value of $AB\bar{C}$.

If two or more paths merge at a node, the fact is expressed by use of a plus sign (+) which means "OR".

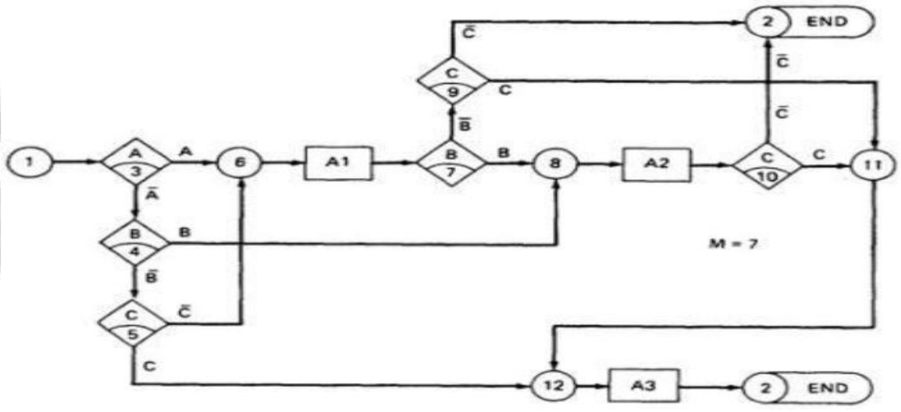


Figure 6.5: A Troublesome Program

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

$$N11 = (N8)C + (N6)\bar{B}C$$
$$N12 = N11 + \bar{A}\bar{B}C$$
$$N2 = N12 + (N8)\bar{C} + (N6)\bar{B}\bar{C}$$

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

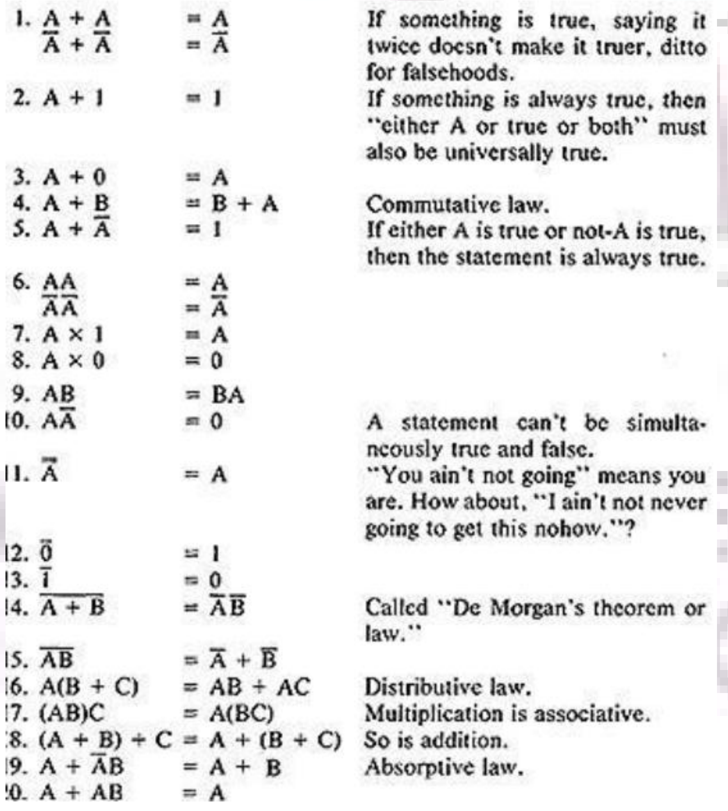
RULES OF BOOLEAN ALGEBRA:
Boolean algebra has three operators: X (AND), + (OR) and $\bar{A}$(NOT)

X : meaning AND. Also called multiplication. A statement such as AB (A X B) 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. This is read as either "not A" or "A

bar". The entire expression under the bar is negated.

The following are the laws of boolean algebra:



1.	$A + A$ $\bar{A} + \bar{A}$	$= A$ $= \bar{A}$	If something is true, saying it twice doesn't make it truer, ditto for falsehoods.
2.	$A + 1$	$= 1$	If something is always true, then "either A or true or both" must also be universally true.
3.	$A + 0$	$= A$	
4.	$A + B$	$= B + A$	Commutative law.
5.	$A + \bar{A}$	$= 1$	If either A is true or not-A is true, then the statement is always true.
6.	AA $\bar{A}\bar{A}$	$= A$ $= \bar{A}$	
7.	$A \times 1$	$= A$	
8.	$A \times 0$	$= 0$	
9.	AB	$= BA$	
10.	$A\bar{A}$	$= 0$	A statement can't be simultaneously true and false.
11.	$\bar{\bar{A}}$	$= A$	"You ain't not going" means you are. How about, "I ain't not never going to get this nohow."?
12.	$\bar{0}$	$= 1$	
13.	$\bar{1}$	$= 0$	
14.	$\overline{A + B}$	$= \bar{A}\bar{B}$	Called "De Morgan's theorem or law."
15.	$\overline{AB}$	$= \bar{A} + \bar{B}$	
16.	$A(B + C)$	$= AB + AC$	Distributive law.
17.	$(AB)C$	$= A(BC)$	Multiplication is associative.
18.	$(A + B) + C$	$= A + (B + C)$	So is addition.
19.	$A + \bar{A}B$	$= A + B$	Absorptive law.
20.	$A + AB$	$= A$	

In all of the above, a letter can represent a single sentence or an entire boolean algebra expression. Individual letters in a boolean algebra expression are called

Literals (e.g. A,B) The product of several literals is called a **product term** (e.g., ABC, DE).

An arbitrary boolean expression that has been multiplied out so that it consists of the sum of products (e.g., ABC + DEF + GH) is said to be in **sum-of-products form**.

The result of simplifications (using the rules above) is again in the sum of product form and each product term in such a simplified version is called a **prime implicant**. For example, ABC + AB DEF reduce by rule 20 to AB + DEF; that is, AB and DEF are prime implicants. The path expressions of Figure 6.5 can now be simplified by applying the rules.

The following are the laws of boolean algebra:

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

Similarly,

$$
\begin{aligned}
N11 &= (N8)C + (N6)\overline{B}C \\
&= BC + (A + \overline{B}\,\overline{C})\overline{B}C && \text{: Substitution.} \\
&= BC + A\overline{B}C && \text{: Rules 16, 9, 10, 8, 3.} \\
&= C(B + \overline{B}A) && \text{: Rules 9, 16.} \\
&= C(B + A) && \text{: Rule 19.} \\
&= AC + BC && \text{: Rules 16, 9, 9, 4.} \\
N12 &= N11 + \overline{A}\,\overline{B}C \\
&= AC + BC + \overline{A}\,\overline{B}C \\
&= C(B + \overline{A}\,\overline{B}) + AC \\
&= C(\overline{A} + B) + AC \\
&= C\overline{A} + AC + BC \\
&= C + BC \\
&= C \\
N2 &= N12 + (N8)\overline{C} + (N6)\overline{B}\,\overline{C} \\
&= C + B\overline{C} + (A + \overline{B}\,\overline{C})\overline{B}\,\overline{C} \\
&= C + B\overline{C} + \overline{B}\,\overline{C} \\
&= C + \overline{C}(B + \overline{B}) \\
&= C + \overline{C} \\
&= 1
\end{aligned}
$$

The deviation from the specification is now clear. The functions should have been:

$$
\begin{aligned}
N6 &= A + \overline{A}\,\overline{B}\,\overline{C} = A + \overline{B}\,\overline{C} && \text{: correct.} \\
N8 &= B + \overline{A}\,\overline{B}\,\overline{C} = B + \overline{A}\,\overline{C} && \text{: wrong, was just B.} \\
N12 &= C + \overline{A}\,\overline{B}\,\overline{C} = C + \overline{A}\,\overline{B} && \text{: wrong, was just C.}
\end{aligned}
$$

Loops complicate things because we may have to solve a boolean equation to determine what predicate value combinations lead to where.

KV CHARTS:

INTRODUCTION:

If you had to deal with expressions in four, five, or six variables, you could get bogged down in the algebra and make as many errors in designing test cases as there are bugs in the routine you're testing.

Karnaugh-Veitch chart reduces boolean algebraic manipulations to graphical trivia.
Beyond six variables these diagrams get cumbersome and may not be effective.

- **SINGLE VARIABLE:**

o Figure 6.6 shows all the boolean functions of a single variable and their equivalent representation as a KV chart.

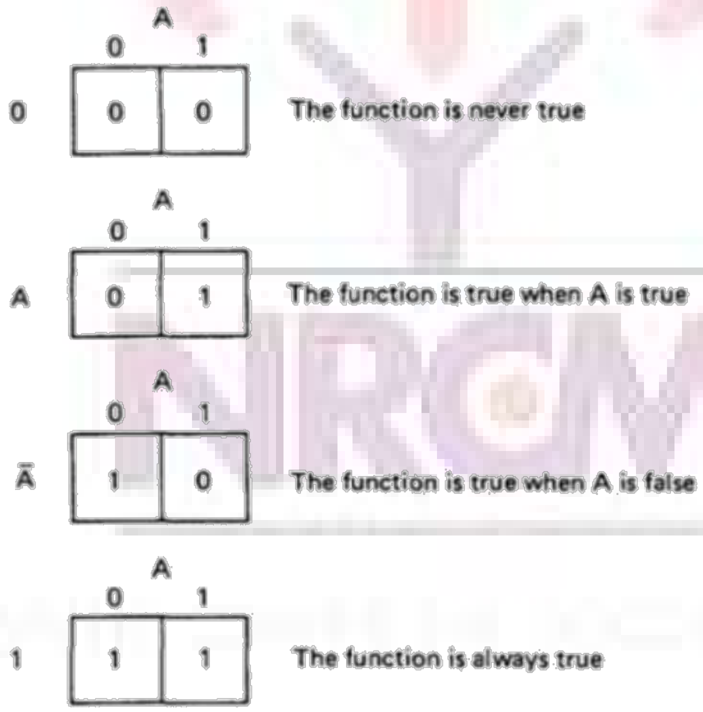

A
0 1
0
0 0
The function is never true
A
A
0 1
The function is true when A is true
$\bar{A}$
1 0
The function is true when A is false
1
1 1
The function is always true

Figure 6.6 : KV Charts for Functions of a Single Variable.

The charts show all possible truth values that the variable A can have.
A "1" means the variable's value is "1" or TRUE. A "0" means that the variable's value is 0 or FALSE.
The entry in the box (0 or 1) specifies whether the function that the chart represents is true or false for that value of the variable.
We usually do not explicitly put in 0 entries but specify only the conditions under which the function is true.

TWO VARIABLES:

Figure 6.7 shows eight of the sixteen possible functions of two variables.

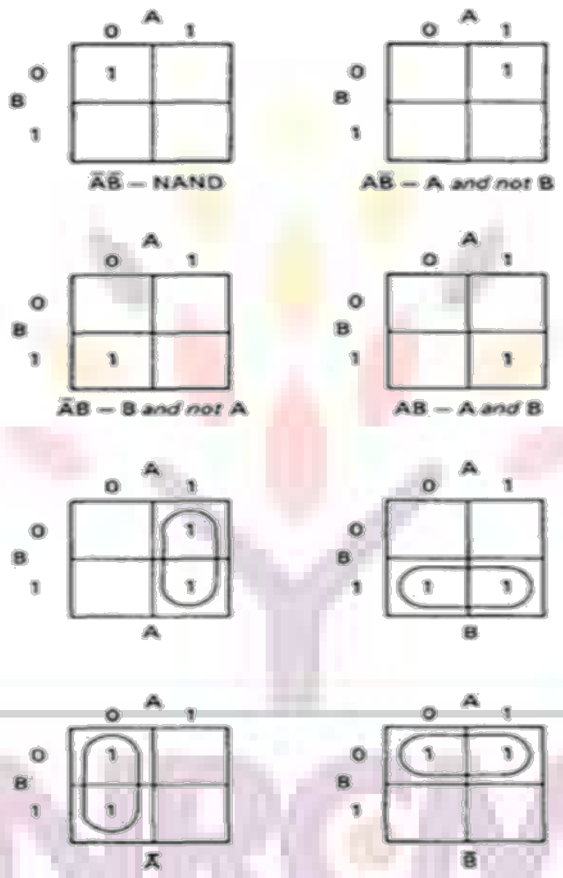


Figure 6.7: KV Charts for Functions of Two Variables.

Each box corresponds to the combination of values of the variables for the row and column of that box.

A pair may be adjacent either horizontally or vertically but not diagonally.
Any variable that changes in either the horizontal or vertical direction does not appear in the expression.

In the fifth chart, the B variable changes from 0 to 1 going down the column, and because the A variable's value for the column is 1, the chart is equivalent to a simple A.

Figure 6.8 shows the remaining eight functions of two variables.

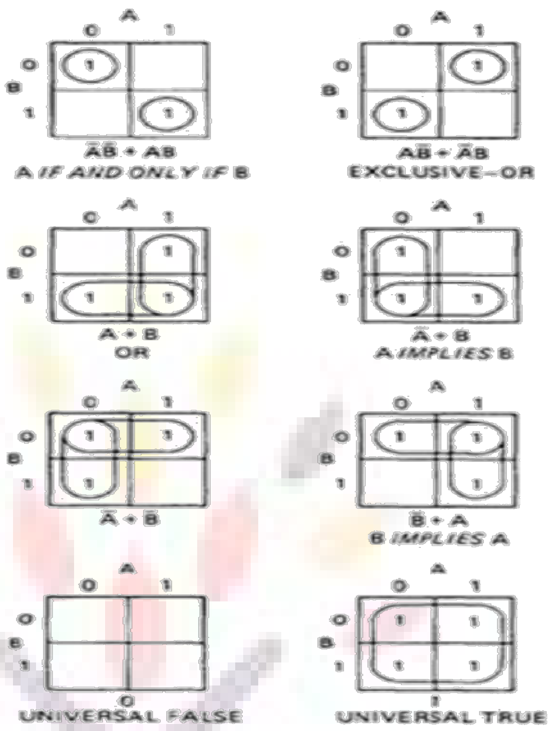
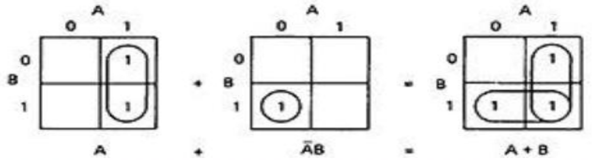


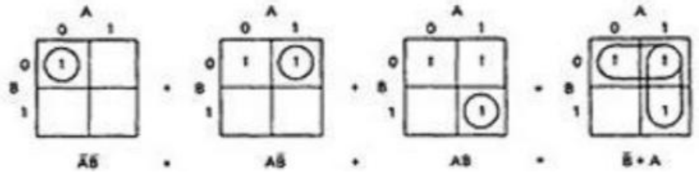
Figure 6.8: More Functions of Two Variables.

The first chart has two 1's in it, but because they are not adjacent, each must be taken separately.
They are written using a plus sign.
It is clear now why there are sixteen functions of two variables.
Each box in the KV chart corresponds to a combination of the variables' values. o That combination might or might not be in the function (i.e., the box corresponding to that combination might have a 1 or 0 entry).

o 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^{2n} ways of doing this.
o Consequently for one variable there are 2^{21} = 4 functions, 16 functions of 2 variables, 256 functions of 3 variables, 16,384 functions of 4 variables, andso on.Given two charts over the same variables, arranged the same way, their product is the term by term product, their sum is the term by term sum, and the negation of a chart is gotten by reversing all the 0 and 1 entries in the chart.



OR



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. o A few examples will illustrate the principles:

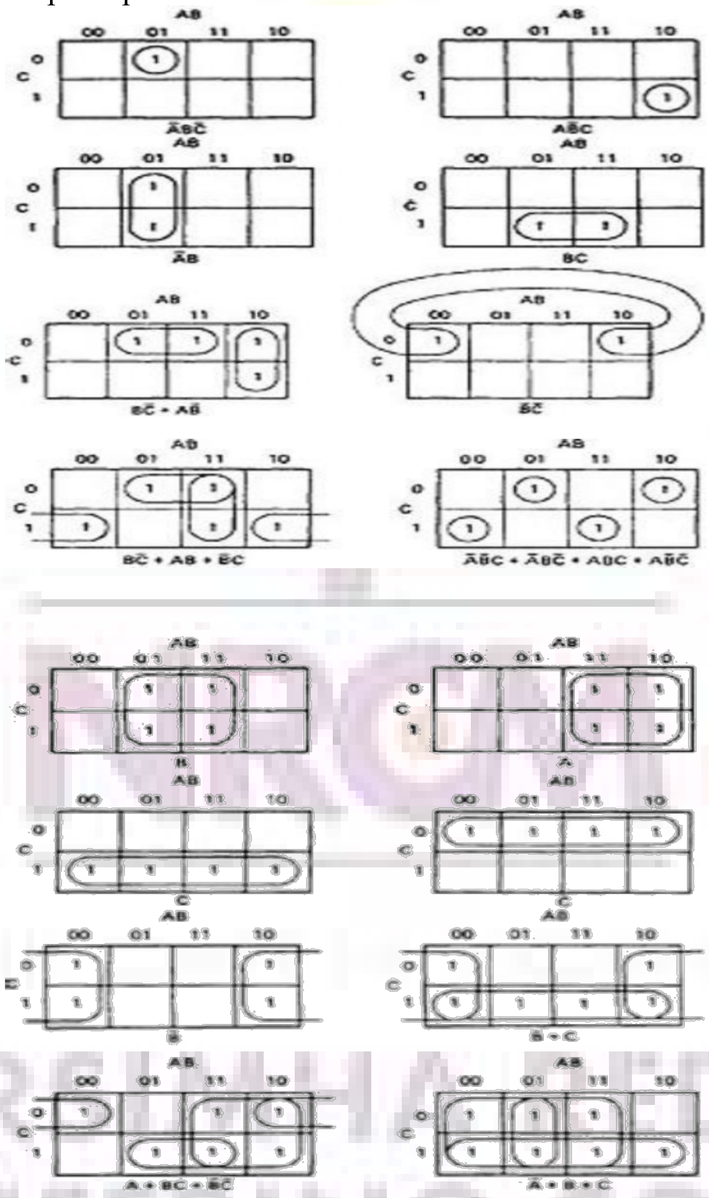


Figure 6.8: KV Charts for Functions of Three Variables.

You'll notice that there are several ways to circle the boxes into maximum-sized covering groups.

<u>**UNIT-IV**</u>

STATES, STATE GRAPHS, AND TRANSITION TESTING

State, State Graphs and Transition testing:- state graphs, good & bad state graphs, state testing, Testability tips.

Introduction

- The finite state machine is as fundamental to software engineering as boolean algebra to logic.
- State testing strategies are based on the use of finite state machine models for software structure, software behavior, or specifications of software behavior.

- Finite state machines can also be implemented as table-driven software, in which case they are a powerful design option.

State Graphs

A state is defined as: “A combination of circumstances or attributes belonging for the time being to a person or thing.”

- For example, a moving automobile whose engine is running can have the following states with respect to its transmission.

Reverse gear
Neutral gear
First gear
Second gear
Third gear

Fourth gear State graph - Example

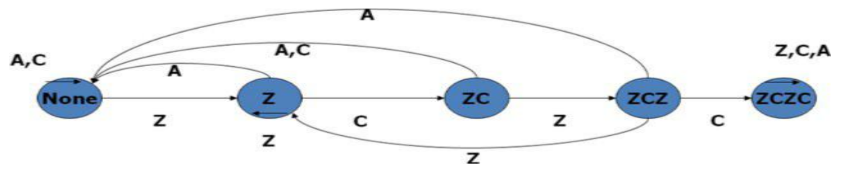
For example, a program that detects the character sequence “ZCZC” can be in the following states.

Neither ZCZC nor any part of it has been detected.

Z has been detected.
ZC has been detected.

ZCZ has been detected.

ZCZC has been detected.



States are represented by Nodes. State are numbered or may identified by words or whatever else is convenient.

Inputs and Transitions

- Whatever is being modeled is subjected to inputs. As a result of those inputs, the state changes, or is said to have made a Transition.
- Transitions are denoted by links that join the states.
- The input that causes the transition are marked on the link; that is, the inputs are link weights.
- There is one out link from every state for every input.

If several inputs in a state cause a transition to the same subsequent state, instead of drawing a bunch of parallel links we can abbreviate the notation by listing the several inputs as in: "input1, input2, input3………".

Finite State Machine

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

Outputs

An output can be associated with any link.

Out puts are denoted by letters or words and are separated from inputs by a slash as follows: "input/output".

As always, output denotes anything of interest that's observable and is not restricted to explicit outputs by devices.

Outputs are also link weights.

If every input associated with a transition causes the same output, then denoted it as: o "input1, input2, input3…………../output"

State tableses

Big state graphs are cluttered and hard to follow.

It's more convenient to represent the state graph as a table (the state table or state transition table) that specifies the states, the inputs, the transitions and the outputs.

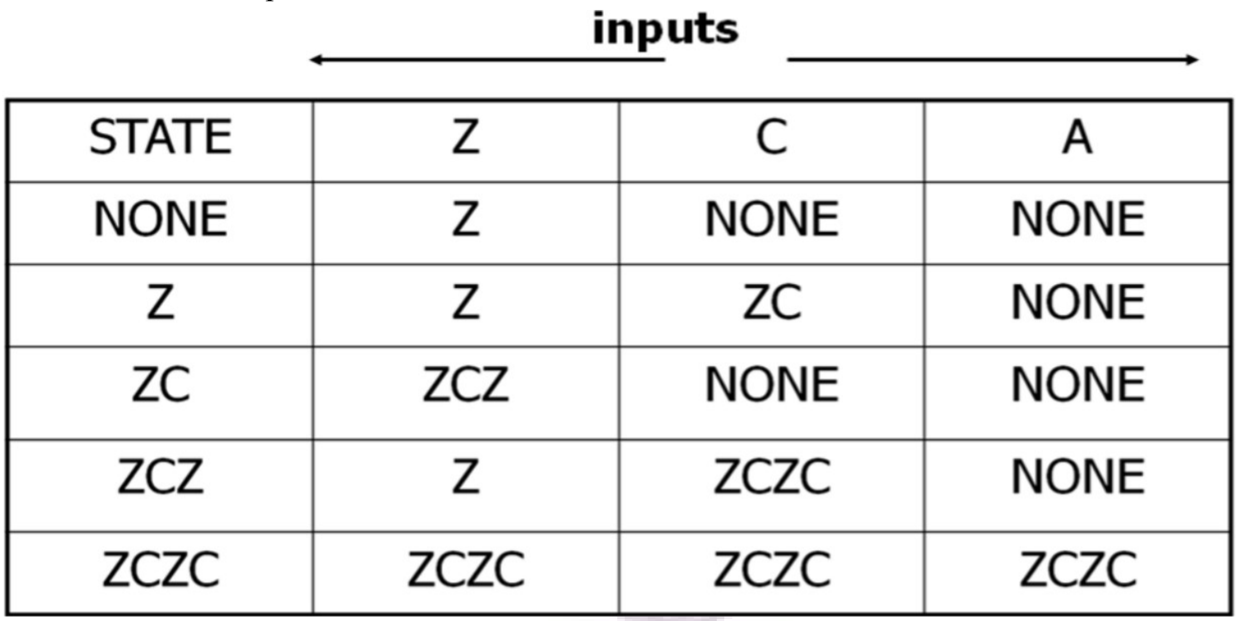
The following conventions are used:

Each row of the table corresponds to a state.

Each column corresponds to an input condition.

The box at the intersection of a row and a column specifies the next state (the transition) and the output, if any.

State Table-Example



	inputs		
STATE	Z	C	A
NONE	Z	NONE	NONE
Z	Z	ZC	NONE
ZC	ZCZ	NONE	NONE
ZCZ	Z	ZCZC	NONE
ZCZC	ZCZC	ZCZC	ZCZC

Time Versus Sequence

State graphs don’t represent time-they represent sequence.

A transition might take microseconds or centuries;

A system could be in one state for milliseconds and another for years- the state graph would be the same because it has no notion of time.

- Although the finite state machines model can be elaborated to include notions of time in addition to sequence, such as time Petri Nets.

Software implementation

- There is rarely a direct correspondence between programs and the behavior of a process described as a state graph.

- The state graph represents, the total behavior consisting of the transport, the software, the executive, the status returns, interrupts, and so on.
- There is no simple correspondence between lines of code and states. The state table forms the basis.

Good State Graphs and Bad

 What constitutes a good or a bad state graph is to some extent biased by the kinds of state graphs that are likely to be used in a software test design context.

Here are some principles for judging.

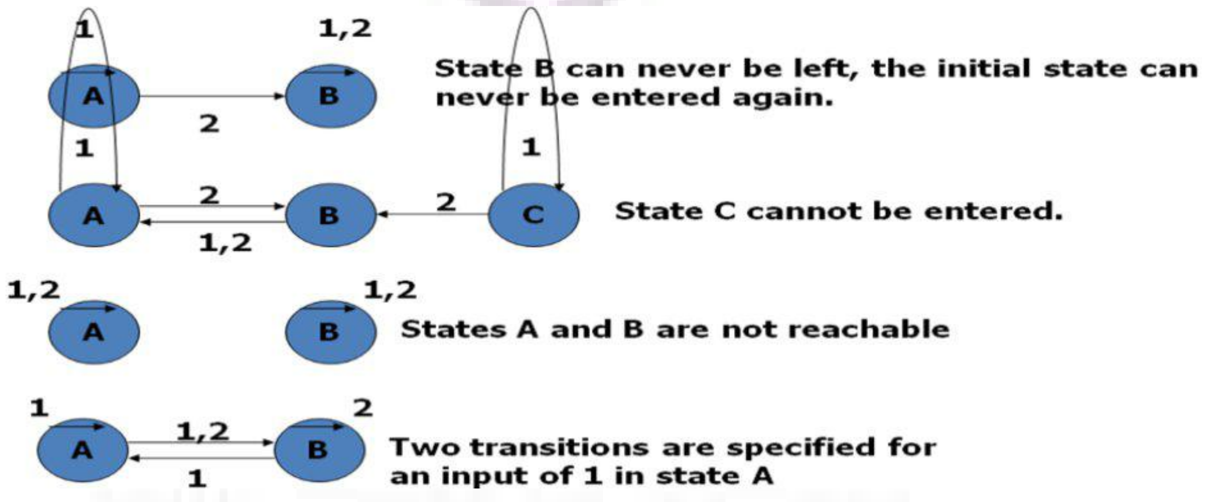
The total number of states is equal to the product of the possibilities of factors that make up the state.

For every state and input there is exactly one transition specified to exactly one, possibly the same, state.

For every transition there is one output action specified. The output could be trivial, but at least one output does something sensible.

For every state there is a sequence of inputs that will drive the system back to the same state.

Important graphs



1
1,2
A
B
2
State B can never be left, the initial state can never be entered again.
1
1
A
2
B
2
C
1,2
State C cannot be entered.
1,2
A
1,2
B
States A and B are not reachable
1
A
1,2
B
2
1
Two transitions are specified for an input of 1 in state A

State Bugs-Number of States
The number of states in a state graph is the number of states we choose to recognize or model.

 The state is directly or indirectly recorded as a combination of values of variables that appear in the data base.

- For example, the state could be composed of the value of a counter whose possible values ranged from 0 to 9, combined with the setting of two bit flags, leading to a total of 2*2*10=40 states.

The number of states can be computed as follows:

Identify all the component factors of the state.
o Identify all the allowable values for each factor.

o The number of states is the product of the number of allowable values of all the factors.

 Before you do anything else, before you consider one test case, discuss the number of states you think there are with the number of states the programmer thinks there are.

There is no point in designing tests intended to check the system's behavior in various states if there's no agreement on how many states there are.

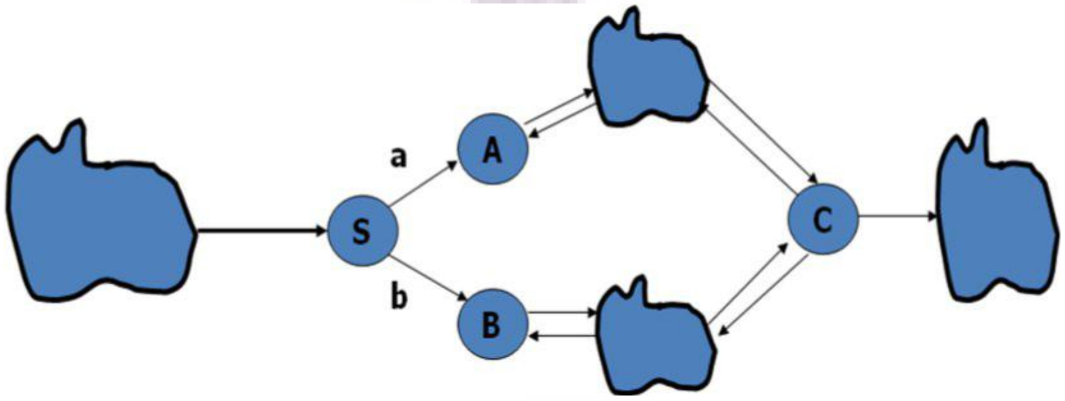
Impossible States

Some times some combinations of factors may appear to beimpossible. The discrepancy between the programmer's state count and the tester's state count is often due to a difference of opinion concerning "impossible states".

 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.

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. This notion can also be extended to set of states.



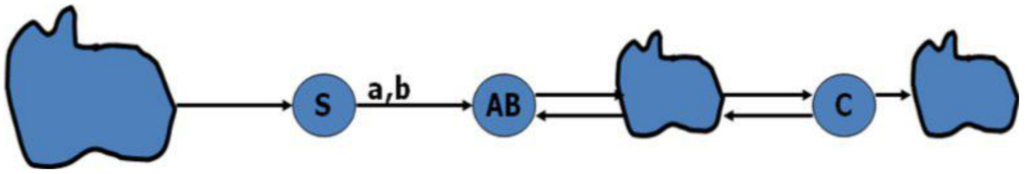
Merging of Equivalent States

Recognizing Equivalent States

Equivalent states can be recognized by the following procedures:

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.

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.



TransitionBugs-

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 the input from occurring in that state.

Exactly one transition must be specified for every combination of input and state.

A program can't have contradictions or ambiguities.

Ambiguities are impossible because the program will do something for every input. Even the state does not change, by definition this is a transition to the same state.

Unreachable States

An unreachable state is like unreachable code.

A state that no input sequence can reach.

An unreachable state is not impossible, just as unreachable code is not impossible There may be transitions from unreachable state to other states; there usually because the state became unreachable as a result of incorrect transition.

There are two possibilities for unreachable states:

There is a bug; that is some transitions are missing.
The transitions are there, but you don't know about it.

Dead States

A dead state is a state that once entered cannot be left.

This is not necessarily a bug but it is suspicious.

Output Errors

The states, transitions, and the inputs could be correct, there could be no dead or unreachable states, but the output for the transition could be incorrect.
Output actions must be verified independently of states and ransitions. State Testing

Impact of Bugs

If a routine is specified as a state graph that has been verified as correct in all details.

Program code or table or a combination of both must still be implemented.

A bug can manifest itself as one of the following symptoms:

Wrong number of states.

Wrong transitions for a given state-input combination.

Wrong output for a given transition.

- Pairs of states or sets of states that are inadvertently made equivalent.

States or set of states that are split to create in equivalent duplicates.

States or sets of states that have become dead.

States or sets of states that have become unreachable.

Principles of State Testing

The strategy for state testing is analogous to that used for path testing flow graphs.

Just as it's impractical to go through every possible path in a flow graph, it's impractical to go through every path in a state graph.

The notion of coverage is identical to that used for flow graphs.

Even though more state testing is done as a single case in a grand tour, it's impractical to do it that way for several reasons.

In the early phases of testing, you will never complete the grand tour because of bugs. Later, in maintenance, testing objectives are understood, and only a few of the states and transitions have to be tested. A grand tour is waste of time.

- Theirs is no much history in a long test sequence and so much has happened that verification is difficult.

Starting point of state testing

- Define a set of covering input sequences that get back to the initial state when starting from the initial state.
- 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:

Input sequences Corresponding transitions or next-state names o Output sequences

Limitations and Extensions

State transition coverage in a state graph model does not guarantee complete testing. How defines a hierarchy of paths and methods for combining paths to produce covers of state graphs.

The simplest is called a “0 switch” which corresponds to testing each transition individually.

The next level consists of testing transitions sequences consisting of two transitions called “1 switches”.

The maximum length switch is “n-1 switch” where there are n numbers of states.Situations at which state testing is useful

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.

Most protocols between systems, between humans and machines, between components of a system. Device drivers such as for tapes and discs that have complicated retry and recovery procedures if the action depends on the state.

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

UNIT-V

GRAPH MATRICES AND APPLICATIONS

Graph Matrices and Application:-Motivational overview, matrix of graph, relations, power of a matrix, node reduction algorithm, building tools. (Student should be given an exposure to a tool like JMeter or Win-runner).

Problem with Pictorial Graphs

Graphs were introduced as an abstraction of software structure.

Whenever a graph is used as a model, sooner or later we trace paths through it- to find a set of covering paths, a set of values that will sensitize paths, the logic function that controls the flow, the processing time of the routine, the equations that define the domain, or whether a state is reachable or not.

Path is not easy, and it's subject to error. You can miss a link here and there or cover some links twice.

One solution to this problem is to represent the graph as a matrix and to use matrix operations equivalent to path tracing. These methods are more methodical and mechanical and don't depend on your ability to see a path they are more reliable.

Tool Building

If you build test tools or want to know how they work, sooner or later you will be implementing or investigating analysis routines based on these methods.

It is hard to build algorithms over visual graphs so the properties or graph matrices are fundamental to tool building.

The Basic Algorithms

The basic tool kit consists of:

Matrix multiplication, which is used to get the path expression from every node to every other node.

A partitioning algorithm for converting graphs with loops into loop free graphs or equivalence classes.
A collapsing process which gets the path expression from any node to any other node.

The Matrix of a Graph

A graph matrix is a square array with one row and one column for every node in the graph.

Each row-column combination corresponds to a relation between the node corresponding to the row and the node corresponding to thecolumn.

The relation for example, could be as simple as the link name, if there is a link between the nodes.

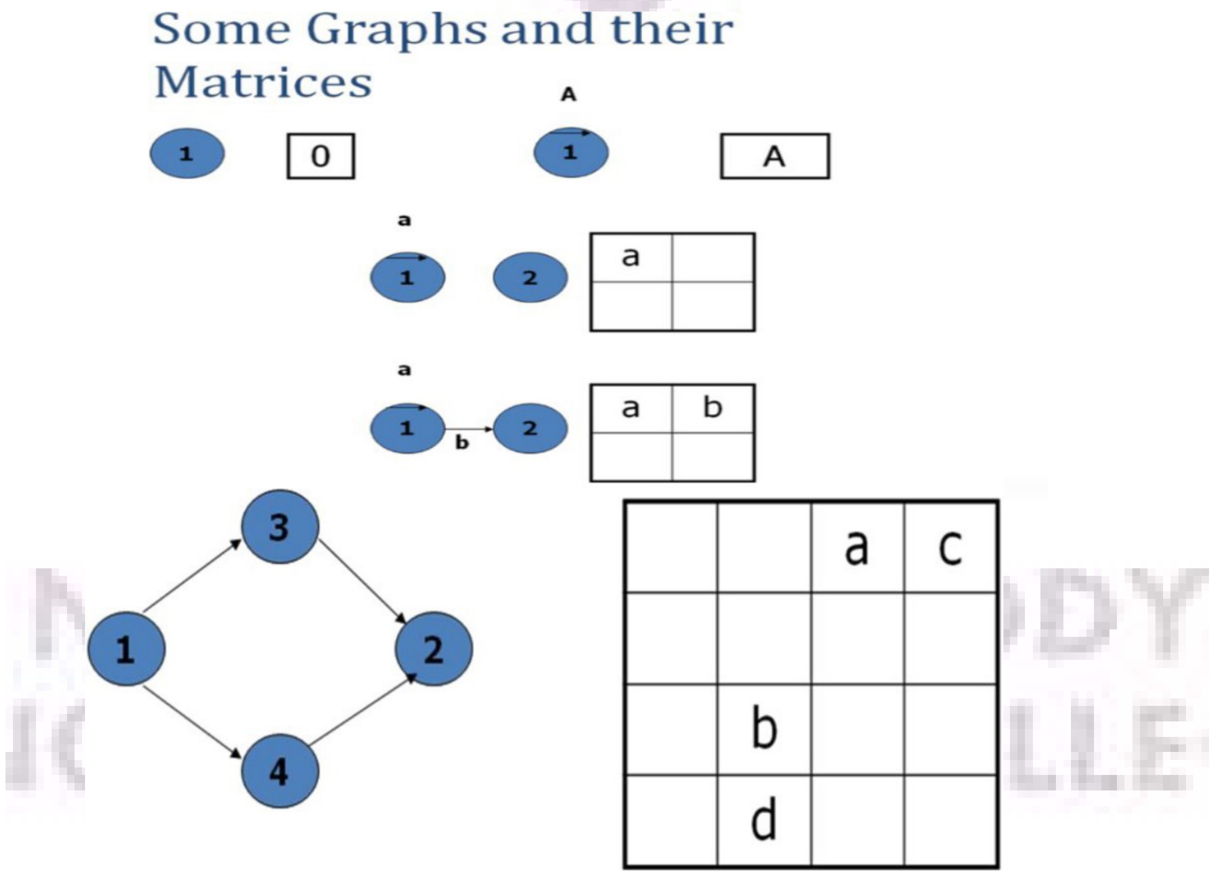
Some of the things to be observed:

The size of the matrix equals the number of nodes.

There is a place to put every possible direct connection or link between any and any other node. The entry at a row and column intersection is the link weight of the link that connects the two nodes in that direction.

A connection from node i to j does not imply a connection from node j to node i.

If there are several links between two nodes, then the entry is a sum; the "+" sign denotes parallel links as usual.



A simple weight

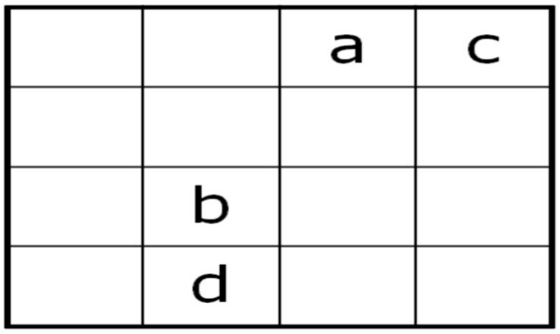
A simplest weight we can use is to note that there is or isn't a connection. Let "1" mean that there is a connection and "0" mean that there isn't.
The arithmetic rules are:

n 1+1= 1*1=1
1+0=
n 1 1*0=0
0+0=
n 0 0*0=0

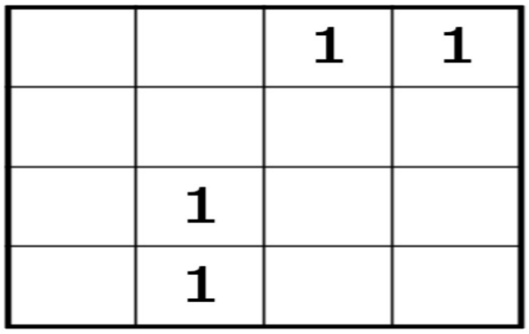
A matrix defined like this is called connection matrix.

Connection matrix

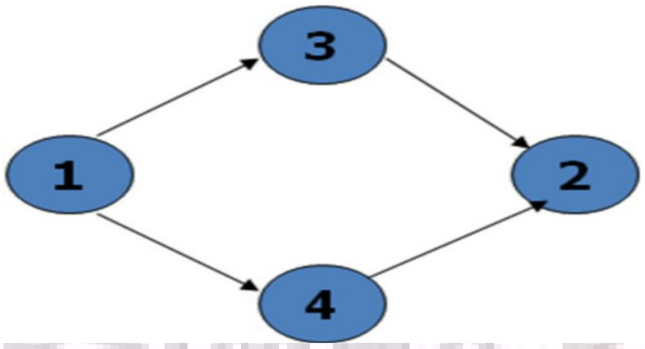
The connection matrix is obtained by replacing each entry with 1 if there is a link and 0 if there isn't.
As usual we don't write down 0 entries to reduce the clutter.

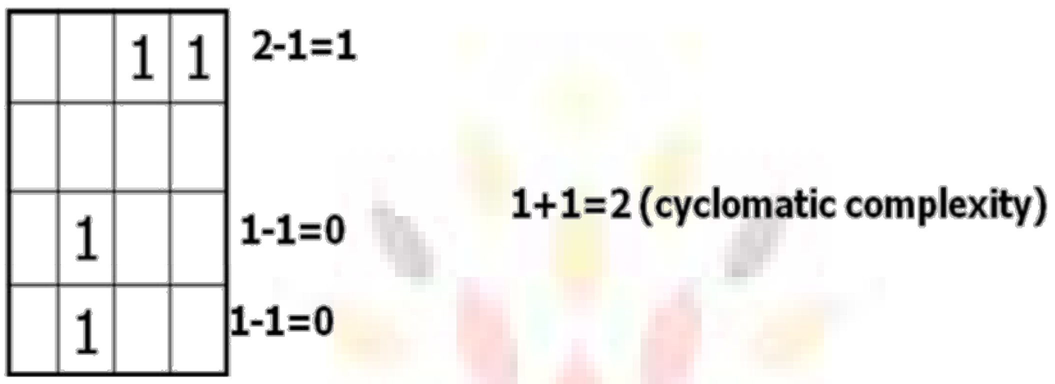


		a	c
	b		
	d		



		1	1
	1		
	1		





		1	1	2-1=1
	1			1-1=0
	1			1-1=0

1+1=2 (cyclomatic complexity)

Connection Matrix-continued

Each row of a matrix denotes the out links of the node corresponding to that row.

Each column denotes the in links corresponding to that node.

A branch is a node with more than one nonzero entry in its row.

A junction is node with more than one nonzero entry in its column.

A self loop is an entry along the diagonal.

Cyclomatic Complexity

The cyclomatic complexity obtained by subtracting 1 from the total number of entries in each row and ignoring rows with no entries, we obtain the equivalent number of decisions for each row. Adding these values and then adding 1 to the sum yields the graph's cyclomatic complexity.

Relations

A relation is a property that exists between two objects of interest.

For example,

“Node a is connected to node b” or aRb where “R” means “is connected to”.

“a>=b” or aRb where “R” means greater than or equal”.

A graph consists of set of abstract objects called nodes and a relation R between the nodes.

If aRb, which is to say that a has the relation R to b, it is denoted by a link from a to b.

For some relations we can associate properties called as link weights.

Transitive Relations

A relation is transitive if aRb and bRc implies aRc.

Most relations used in testing are transitive.

Examples of transitive relations include: is connected to, is greater than or equal to, is less than or equal to, is a relative of, is faster than, is slower than, takes more time than, is a subset of, includes, shadows, is the boss of.

Examples of intransitive relations include: is acquainted with, is a friend of, is a neighbor of, is lied to, has a du chain between.

Reflexive Relations

A relation R is reflexive if, for every a, aRa.

A reflexive relation is equivalent to a self loop at every node.

Examples of reflexive relations include: equals, is acquainted with, is a relative of.

Examples of irreflexive relations include: not equals, is a friend of, is on top of, is under.

Symmetric Relations

A relation R is symmetric if for every a and b, aRb implies bRa.

A symmetric relation mean that if there is a link from a to b then there is also a link from b to a. A graph whose relations are not symmetric are called directed graph.

A graph over a symmetric relation is called an undirected graph.

The matrix of an undirected graph is symmetric (aij=aji) for all i,j)

Antisymmetric Relations

A relation R is antisymmetric if for every a and b, if aRb and bRa, then a=b, or they are the same elements.

Examples of antisymmetric relations: is greater than or equal to, is a subset of, time.

Examples of nonantisymmetric relations: is connected to, can be reached from, is greater than, is a relative of, is a friend of

Equivalence Relations

An equivalence relation is a relation that satisfies the reflexive, transitive, and symmetric properties.

Equality is the most familiar example of an equivalence relation.

If a set of objects satisfy an equivalence relation, we say that they form an equivalence class over that relation.

The importance of equivalence classes and relations is that any member of the equivalence class is, with respect to the relation, equivalent to any other member of that class.

The idea behind partition testing strategies such as domain testing and path testing, is that we can partition the input space into equivalence classes.

Testing any member of the equivalence class is as effective as testing them all.

Partial Ordering Relations

A partial ordering relation satisfies the reflexive, transitive, and antisymmetric properties.

Partial ordered graphs have several important properties: they are loop free, there is at least one maximum element, and there is at least one minimum element.

The Powers of a Matrix

Each entry in the graph’s matrix expresses a relation between the pair of nodes that corresponds to that entry.

Squaring the matrix yields a new matrix that expresses the relation between each pair of nodes via one intermediate node under the assumption that the relation istransitive.

The square of the matrix represents all path segments two links long.

The third power represents all path segments three links long.

Matrix Powers and Products

Given a matrix whose entries are aij, the square of that matrix is obtained by replacing every entry with

$$a_{ij}=\sum_{k=1}^{n} a_{ik}\, a_{kj}$$

more generally, given two matrices A and B with entries aik and bkj, respectively, their product is a new matrix C, whose entries are cij, where:

$$C_{ij}=\sum_{k=1}^{n} a_{ik}\, b_{kj}$$

Partitioning Algorithm

Consider any graph over a transitive relation. The graph may have loops.

We would like to partition the graph by grouping nodes in such a way that every loop is contained within one group or another.

Such a graph is partially ordered.

There are many used for an algorithm that does that:

We might want to embed the loops within a subroutine so as to have a resulting graph which is loop free at the top level.

Many graphs with loops are easy to analyze if you know where to break the loops.

While you and I can recognize loops, it’s much harder to program a tool to do it unless you have a solid algorithm on which to base the tool.

Node Reduction Algorithm (General)

The matrix powers usually tell us more than we want to know about most graphs.

In the context of testing, we usually interested in establishing a relation between two nodes-typically the entry and exit nodes.

In a debugging context it is unlikely that we would want to know the path expression between every node and every other node.

The advantage of matrix reduction method is that it is more methodical than the graphical method called as node by node removal algorithm.

Select a node for removal; replace the node by equivalent links that bypass that node and add those links to the links they parallel.

Combine the parallel terms and simplify as you can.

Observe loop terms and adjust the out links of every node that had a self loop to account for the effect of the loop.

The result is a matrix whose size has been reduced by 1. Continue until only the two nodes of interest exist.