MAR GREGORIOS COLLEGE OF ARTS & SCIENCE

Block No.8, College Road, Mogappair West, Chennai – 37

Affiliated to the University of Madras
Approved by the Government of Tamil Nadu
An ISO 9001:2015 Certified Institution

DEPARTMENT OF COMPUTER APPLICATION

SUBJECT NAME: SOFTWARE TESTING

SUBJECT CODE: SAZ6C

SEMESTER: VI

PREPARED BY: PROF. K.RAJALAKSHMI

SOFTWARE TESTING

SYLLABUS

Unit-1: Introduction: Purpose – Productivity and Quality in Software – Testing Vs Debugging – Model for Testing – Bugs – Types of Bugs – Testing and Design Style.

Unit-2: Flow/Graphs and Path Testing – Achievable paths – Path instrumentation – Application – Transaction Flow Testing Techniques

Unit-3: Data Flow Testing Strategies - Domain Testing: Domains and Paths – Domains and Interface Testing .

Unit-4: Linguistic –Metrics – Structural Metric – Path Products and Path Expressions. Syntax Testing – Formats – Test Cases .

Unit-5 : Logic Based Testing – Decision Tables – Transition Testing – States, State Graph, State Testing.

1. Recommended Texts
   i. B. Beizer , 2003, Software Testing Techniques, II Edn., DreamTech India, New Delhi.
   ii. K.V.KK. Prasad , 2005, Software Testing Tools, DreamTech. India, New Delhi. 2.
2. Reference Books
   i. Burnstein, 2003, Practical Software Testing, Springer International Edn.
   ii. E. Kit, 1995, Software Testing in the Real World: Improving the Process, Pearson Education, Delhi.
   iii. R.Rajani, and P.P.Oak, 2004, Software Testing, Tata Mcgraw Hill, New Delhi.

SOFTWARE TESTING

UNIT I

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.

Definition of Software Testing

Software testing can be stated as the process of verifying and validating that a software or application is bug free, meets the technical requirements as guided by its design and development and meets the user requirements effectively and efficiently with handling all the exceptional and boundary cases.

The process of software testing aims not only at finding faults in the existing software but also at finding measures to improve the software in terms of efficiency, accuracy and usability. It mainly aims at measuring specification, functionality and performance of a software program or application.

Software testing can be divided into two steps:
1. **Verification:** it refers to the set of tasks that ensure that software correctly implements a specific function.
2. **Validation:** it refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
Verification: –Are we building the product right?‖
Validation: –Are we building the right product?‖

What are different techniques of Software Testing?

Software techniques can be majorly classified into two categories:

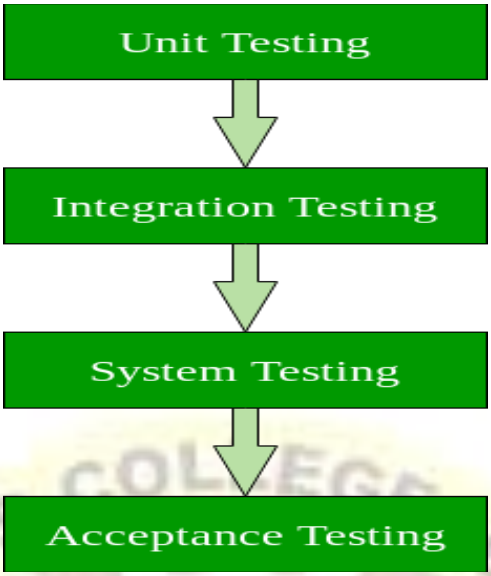
1. **Black Box Testing:** The technique of testing in which the tester doesn't have access to the source code of the software and is conducted at the software interface without concerning with the internal logical structure of the software is known as black box testing.
2. **White-Box Testing:** The technique of testing in which the tester is aware of the internal workings of the product, have access to it's source code and is conducted by making sure that all internal operations are performed according to the specifications is known as white box testing.

Black Box Testing	White Box Testing
Internal workings of an application are not required.	Knowledge of the internal workings is must.
Also known as closed box/data driven testing.	Also known as clear box/structural testing.
End users, testers and developers.	Normally done by testers and developers.
This can only be done by trial and error method.	Data domains and internal boundaries can be better tested.

What are different levels of software testing?
Software level testing can be majorly classified into 4 levels:

1. **Unit Testing:** A level of the software testing process where individual units/components of a software/system are tested. The purpose is to validate that each unit of the software performs as designed.
2. **Integration Testing:** A level of the software testing process where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.

3. **System Testing:** A level of the software testing process where a complete, integrated system/software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.
4. **Acceptance Testing:** A level of the software testing process where a system is tested for acceptability. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.



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. (Its 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.*

- **<u>Phases in a tester's mental life can be categorised into the following 5 phases:</u>**

  - **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'll find a bug.
  - **Phase 2: (1979-1982: Destruction Oriented)** The purpose of testing is to show that software doesnt 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 labour of testing. Other reason is to check the testable and non-testable code. Testable code has fewer bugs than the code that's hard to test. Identifying the testing techniques to test the code is the main key here.

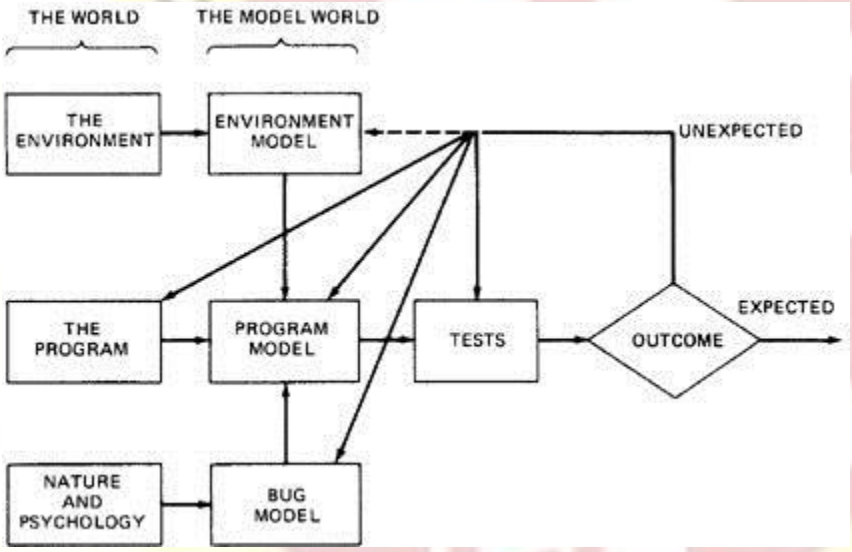
- **Test Design:** We know that the software code must be designed and tested, but many appear to be unaware that tests themselves must be designed and tested. Tests should be properly designed and tested before applying it to the acutal code.
- *Testing is'nt everything:* There are approaches other than testing to create better software. Methods other than testing include:
    - **Inspection Methods:** Methods like walkthroughs, deskchecking, formal inspections and code reading appear to be as effective as testing but the bugs caught donot completely overlap.
    - **Design Style:** While designing the software itself, adopting stylistic objectives such as testability, openness and clarity can do much to prevent bugs.
    - **Static Analysis Methods:** Includes formal analysis of source code during compilation. In earlier days, it is a routine job of the programmer to do that. Now, the compilers have taken over that job.
    - **Languages:** The source language can help reduce certain kinds of bugs. Programmers find new bugs while using new languages.
    - **Development Methodologies and Development Environment:** The development process and the environment in which that methodology is embedded can prevent many kinds of bugs.

- **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 intial conditions and the end can not 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 intutive 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.

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 inorder to test it.
  - If simple model of the program doesnot explain the unexpected behaviour, we may have to modify that model to include more facts and details. And if that fails, we may have to modify the program.

- **BUGS:**
  - Bugs are more insidious (deceiving but harmful) than ever we expect them to be.
  - An unexpected test result may lead us to change our notion of what a bug is and our model of bugs.
  - Some optimistic notions that many programmers or testers have about bugs are usually unable to test effectively and unable to justify the dirty tests most programs need.
- **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 effects only that component's behaviour.
  - **Control Bug Dominance:** The belief that errors in the control structures (if, switch etc) of programs dominate the bugs.
  - **Code / Data Separation:** The belief that bugs respect the separation of code and data.
  - **Lingua Salvator Est:** The belief that the language syntax and semantics (e.g. Structured Coding, Strong typing, etc) eliminates most bugs.
  - **Corrections Abide:** The mistaken belief that a corrected bug remains corrected.
  - **Silver Bullets:** The mistaken belief that X (Language, Design method, representation, environment) grants immunity from bugs.
  - **Sadism Suffices:** The common belief (especially by independent tester) that a sadistic streak, low cunning, and intuition are sufficient to eliminate most bugs. Tough bugs need methodology and techniques.
  - **Angelic Testers:** The belief that testers are better at test design than programmers are at code design.

- **IS COMPLETE TESTING POSSIBLE?**
  - If the objective of the testing were to prove that a program is free of bugs, then testing not only would be practically impossible, but also would be theoretically impossible.
  - **Three different approaches can be used to demonstrate that a program is correct.They are:**
    1. **Functional Testing:**
       - Every program operates on a finite number of inputs. A complete functional test would consists of subjecting the program to all possible input streams.
       - For each input the routine either accepts the stream and produces a correct outcome, accepts the stream and produces an incorrect outcome, or rejects the stream and tells us that it did so.
       - For example, a 10 character input string has 280 possible input streams and corresponding outcomes, so complete functional testing in this sense is IMPRACTICAL.
       - But even theoritically, we can't execute a purely functional test this way because we don't know the length of the string to which the system is responding.
    2. **Structural Testing:**
       - The design should have enough tests to ensure that every path through the routine is exercised at least once. Right off that's is impossible because some loops might never terminate.
       - The number of paths through a small routine can be awesome because each loop multiplies the path count by the number of times through the loop.
       - A small routine can have millions or billions of paths, so total **Path Testing** is usually IMPRACTICAL.

3. Formal Proofs of Correctness:

- Formal proofs of correctness rely on a combination of functional and structural concepts.
- Requirements are stated in a formal language (e.g. Mathematics) and each program statement is examined and used in a step of an inductive proof that the routine will produce the correct outcome for all possible input sequences.
- The IMPRACTICAL thing here is that such proofs are very expensive and have been applied only to numerical routines or to formal proofs for crucial software such as system's security kernel or portions of compilers.

- Each approach leads to the conclusion that complete testing, in the sense of a proof is neither theoretically nor practically possible.

- **IMPORTANCE OF BUGS:** The importance of bugs depends on frequency, correction cost, installation cost, and consequences.
  - **Frequency:** How often does that kind of bug occur? Pay more attention to the more frequent bug types.
  - **Correction Cost:** What does it cost to correct the bug after it is found? The cost is the sum of 2 factors: (1) the cost of discovery (2) the cost of correction. These costs go up dramatically later in the development cycle when the bug is discovered. Correction cost also depends on system size.
  - **Installation Cost:** Installation cost depends on the number of installations: small for a single user program but more for distributed systems. Fixing one bug and distributing the fix could exceed the entire system's development cost.
  - **Consequences:** What are the consequences of the bug? Bug consequences can range from mild to catastrophic.

A reasonable metric for bug importance is

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

- **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:
    1. **Mild:** The symptoms of the bug offend us aesthetically (gently); a misspelled output or a misaligned printout.
    2. **Moderate:** Outputs are misleading or redundant. The bug impacts the system's performance.
    3. **Annoying:** The system's behaviour because of the bug is dehumanizing. *E.g.* Names are truncated orarbitarily modified.
    4. **Disturbing:** It refuses to handle legitimate (authorized / legal) transactions. The ATM wont give you money. My credit card is declared invalid.
    5. **Serious:** It loses track of its transactions. Not just the transaction itself but the fact that the transaction occurred. Accountability is lost.
    6. **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.
    7. **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.
    8. **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.
    9. **Catastrophic:** The decision to shut down is taken out of our hands because the system fails.
    10. **Infectious:** What can be worse than a failed system? One that corrupt other systems even though it doesnot fall in itself ; that erodes the social physical environment; that melts nuclear reactors and starts war.

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 Functionlity

bugs include:

1. **Requirements and Specifications Bugs:**
   - Requirements and specifications developed from them can be incomplete ambiguous, or self-contradictory. They can be misunderstood or impossible to understand.
   - The specifications that don't have flaws in them may change while the design is in progress. The features are added, modified and deleted.
   - Requirements, especially, as expressed in specifications are a major source of expensive bugs.
   - The range is from a few percentage to more than 50%, depending on the application and environment.
   - What hurts most about the bugs is that they are the earliest to invade the system and the last to leave.
2. **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.
3. **Feature Interaction Bugs:**
   - Providing correct, clear, implementable and testable feature specifications is not enough.
   - Features usually come in groups or related features. The features of each group and the interaction of features with in the group are usually well tested.
   - The problem is unpredictable interactions between feature groups or even between individual features. For example, your telephone is provided with call holding and call forwarding. The interactions between these two features may have bugs.
   - Every application has its peculiar set of features and a much bigger set of unspecified feature interaction potentials and therefore result in feature interaction bugs.

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 testsand consistent test criteria.
- The specification problem has been shifted to a higher level but not eliminated.

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

- **STRUCTURAL BUGS:** Various categories in Structural bugs include:
  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, sphagetti code, and worst of all, pachinko code.
     - One reason for control flow bugs is that this area is amenable (supportive) to theoritical treatment.
     - Most of the control flow bugs are easily tested and caught in unit testing.
     - Another reason for control flow bugs is that use of old code especially ALP & COBOL code are dominated by control flow bugs.
     - Control and sequence bugs at all levels are caught by testing, especially structural testing, more specifically path testing combined with a bottom line functional test based on a specification.

2. **Logic Bugs:**
   - Bugs in logic, especially those related to misundertanding 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.
3. **Processing Bugs:**
   - Processing bugs include arithmetic bugs, algebraic, mathematical function evaluation, algorithm selection and general processing.
   - Examples of Processing bugs include: Incorrect conversion from one data representation to other, ignoring overflow, improper use of grater-than-or-eual etc
   - Although these bugs are frequent (12%), they tend to be caught in good unit testing.
4. **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.
5. **Data-Flow Bugs and Anomalies:**
   - Most initialization bugs are special case of data flow anamolies.
   - A data flow anomaly occurs where there is a path along which we expect to do something unreasonable with data, such as using an uninitialized variable, attempting to use a variable before it exists, modifying and then not storing or using the result, or initializing twice without an intermediate use.

- **DATA BUGS:**
  1. Data bugs include all bugs that arise from the specification of data objects, their formats, the number of such objects, and their initial values.
  2. Data Bugs are atleast as common as bugs in code, but they are foten treated as if they didnot exist at all.
  3. *Code migrates data:* Software is evolving towards programs in which more and more of the control and processing functions are stored in tables.
  4. 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.
  5. **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.
  6. **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.
  7. **Content, Structure and Attributes: Content** can be an actual bit pattern, character string, or number put into a data structure. Content is a pure bit pattern and has no meaning unless it is interpreted by a hardware or software processor. All data bugs result in the corruption or misinterpretation of content. **Structure** relates to the size, shape and numbers that describe the data object, that is memory location used to store the content. (e.g A two dimensional array). **Attributes** relates to the specification meaning that is the semantics associated with the contents of a data object. (e.g. an integer, an alphanumeric string, a subroutine). *The severity and subtlelty of bugs increases*

as we go from content to attributes because the things get less formal in that direction.

- **CODING BUGS:**
  1. Coding errors of all kinds can create any of the other kind of bugs.
  2. Syntax errors are generally not important in the scheme of things if the source language translator has adequate syntax checking.
  3. If a program has many syntax errors, then we should expect many logic and coding bugs.
  4. 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.
  2. **Internal Interfaces:**
     - Internal interfaces are in principle not different from external interfaces but they are more controlled.
     - A best example for internal interfaces are communicating routines.
     - The external environment is fixed and the system must adapt to it but the internal environment, which consists of interfaces with other components, can be negotiated.
     - Internal interfaces have the same problem as external interfaces.

3. **Hardware Architecture:**
    - Bugs related to hardware architecture originate mostly from misunderstanding how the hardware works.
    - Examples of hardware architecture bugs: address generation error, i/o device operation / instruction error, waiting too long for a response, incorrect interrupt handling etc.
    - The remedy for hardware architecture and interface problems is two fold: (1) Good Programming and Testing (2) Centralization of hardware interface software in programs written by hardware interface specialists.
4. **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.
5. **Software Architecture:**
    - Software architecture bugs are the kind that called - interactive.
    - Routines can pass unit and integration testing without revealing such bugs.
    - Many of them depend on load, and their symptoms emerge only when the system is stressed.
    - Sample for such bugs: Assumption that there will be no interrupts, Failure to block or un block interrupts, Assumption that memory and registers were initialized or not initialized etc
    - Careful integration of modules and subjecting the final system to a stress test are effective methods for these bugs.
6. **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.

7. 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.

8. Integration Bugs:

- Integration bugs are bugs having to do with the integration of, and with the interfaces between, working and tested components.
- These bugs results from inconsistencies or incompatibilities between components.
- The communication methods include data structures, call sequences, registers, semaphores, communication links and protocols results in integration bugs.
- The integration bugs do not constitute a big bug category(9%) they are expensive category because they are usually caught late in the game and because they force changes in several components and/or data structures.

9. 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:
    1. **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 donot have to make concessions to efficiency.
    2. **Test Quality Assurance:** Programmers have the right to ask how quality in independent testing is monitored.
    3. **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.
    4. **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.

UNIT-II

FLOWGRAPHS AND PATH TESTING:

This unit gives an in depth overview of path testing and its applications.

At the end of this unit, the student will be able to:

- Understand the concept of path testing.
- Identify the components of a control flow diagram and compare the same with a flowchart.
- Represent the control flow graph in the form of a Linked List notation.
- Understand the path testing and selection criteria and their limitations.
- Interpret a control flow-graph and demonstrate the complete path testing to achieve C1+C2.
- Classify the predicates and variables as dependent/independent and correlated/uncorrelated.
- Understand the path sensitizing method and classify whether the path is achievable or not.
- Identify the problem due to co-incidental correctness and choose a path instrumentation method to overcome the problem.

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.

(1) Process Block (2) Decisions (3) Junctions (4) Case Statements

1. Process Block:

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

It is a sequence of statements such that if any one of statement of the block is executed, then all statement thereof is executed.

Formally, a process block is a piece of straight line code of one statement or hundreds of statements.

A process has one entry and one exit. It can consist of a single statement or instruction, a sequence of statements or instructions, a single entry/exit subroutine, a macro or function call, or a sequence of these.

2. Decisions:

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

Machine language conditional branch and conditional skip instructions are examples of decisions.

Most of the decisions are two-way but some are three way branches in control flow.

3. Case Statements:

A case statement is a multi-way branch or decisions.

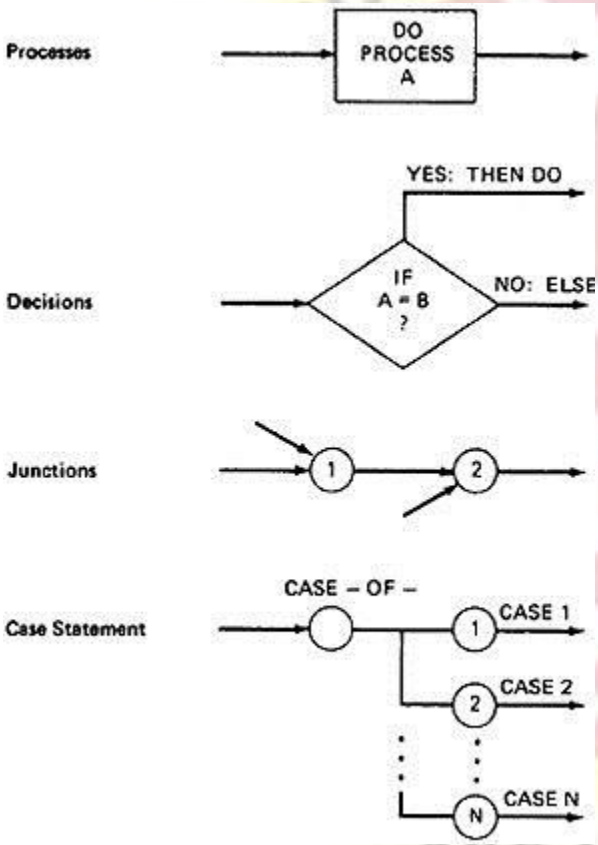
Examples of case statement are a jump table in assembly language, and the PASCAL case statement.

From the point of view of test design, there are no differences between Decisions and Case Statements

4. Junctions:

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

Examples of junctions are: the target of a jump or skip instruction in ALP, a label that is a target of GOTO.



CONTROL FLOW GRAPHS Vs. FLOWCHARTS:

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

NOTATIONAL EVOULTION:

The control flow graph is simplified representation of the program's structure. The notation changes made in creation of control flow graphs:

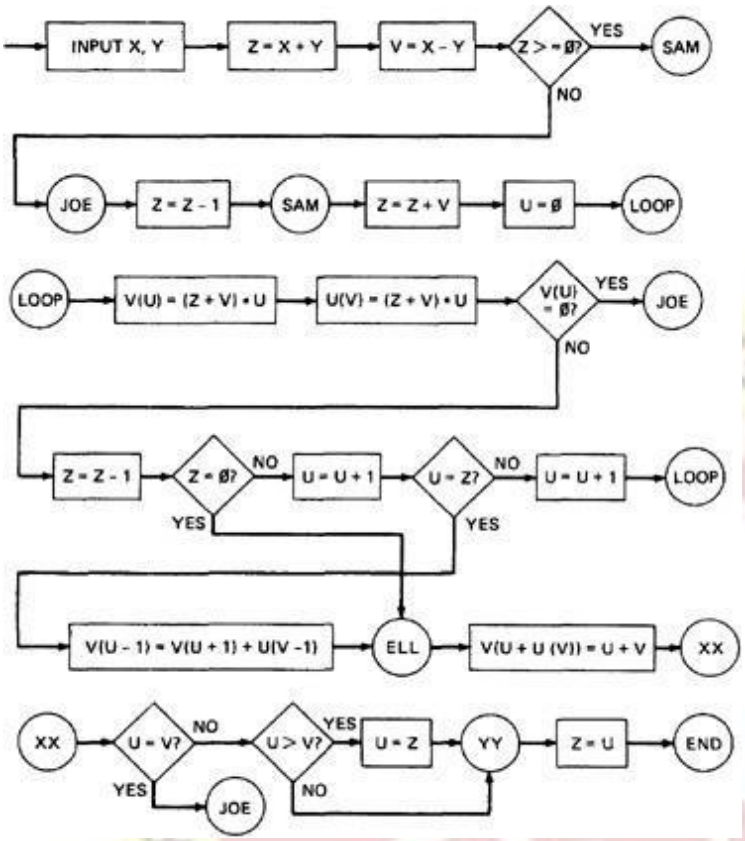
- The process boxes weren't really needed. There is an implied process on every line joining junctions and decisions.
- We don't need to know the specifics of the decisions, just the fact that there is a branch.
- The specific target label names aren't important-just the fact that they exist. So we can replace them by simple numbers.
- To understand this, we will go through an example written in a FORTRAN like programming language called **Programming Design Language (PDL)**. The program's corresponding flowchart
- The first step in translating the program to a flowchart is shown in, 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.

CODE* (PDL)

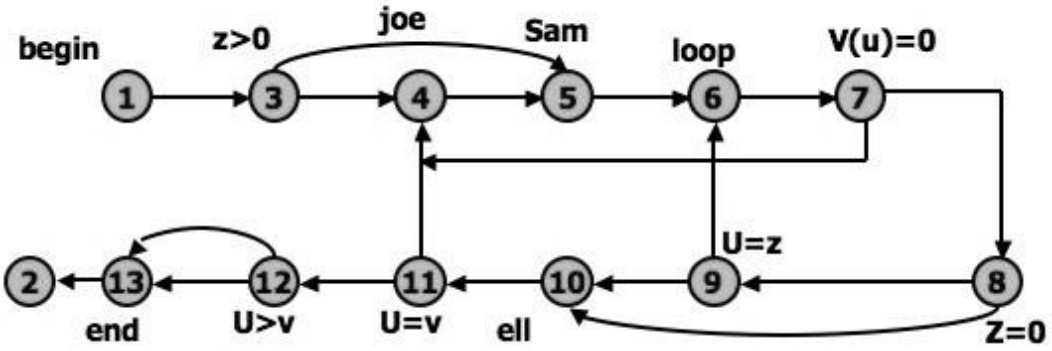
```
      INPUT X, Y                    V(U−1):=V(U+1) + U(V−1)
      Z := X + Y                ELL:V(U+U(V)) := U + V
      V := X − Y                    IF U = V GOTO JOE
      IF Z >=Ø GOTO SAM             IF U > V THEN U := Z
JOE:  Z := Z − 1                    Z := U
SAM:  Z := Z + V           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

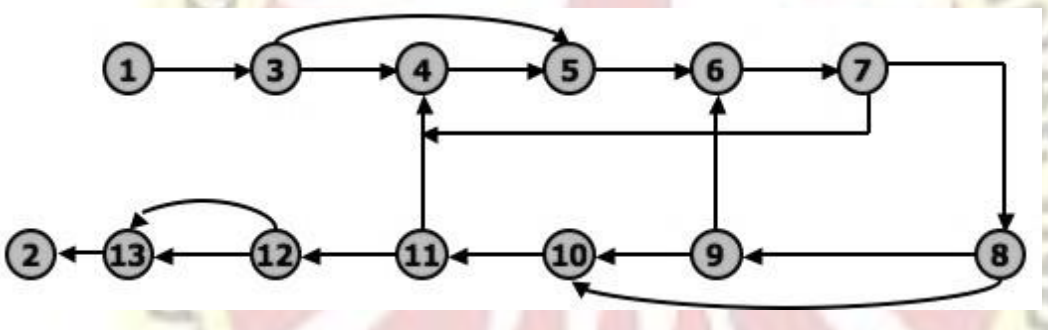
Flowchart for the above PDL



Simplified Flowgraph Notation



Even Simplified Flowgraph Notation



LINKED LIST REPRESENTATION:

Although graphical representations of flowgraphs 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)
```

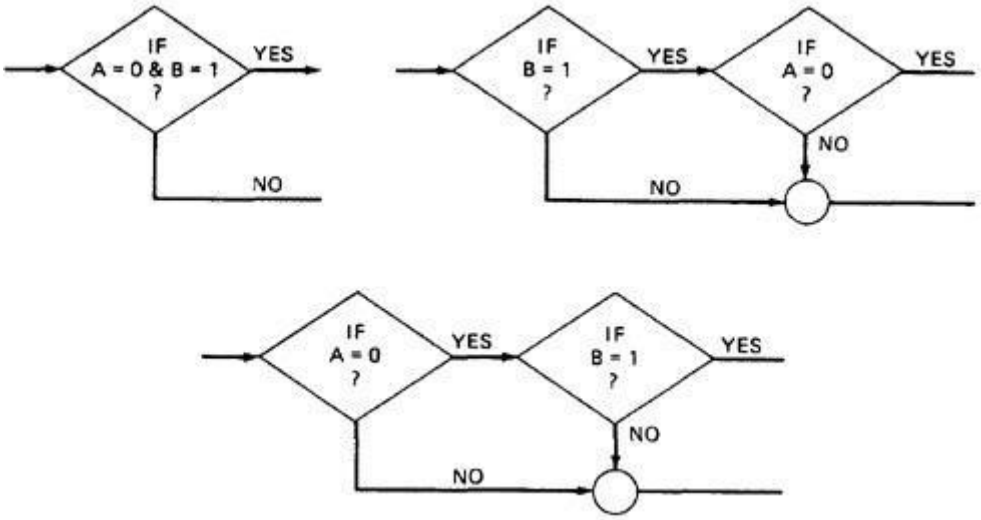
FLOWGRAPH - PROGRAM CORRESPONDENCE:

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.

Alternative Flowgraphs 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
0. Handwritten by the programmer.
1. Automatically produced by a flowcharting program based on a mechanical analysis of the source code.
2. 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 same junction, 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.

The length of path measured by the number of links in it and not by the number of the instructions or statements executed along that path.

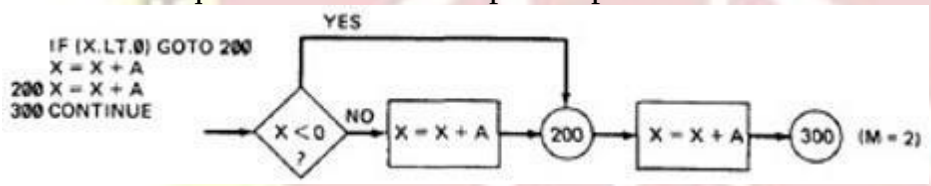
The name of a path is the name of the nodes along the path.

FUNDAMENTAL PATH SELECTION CRITERIA:

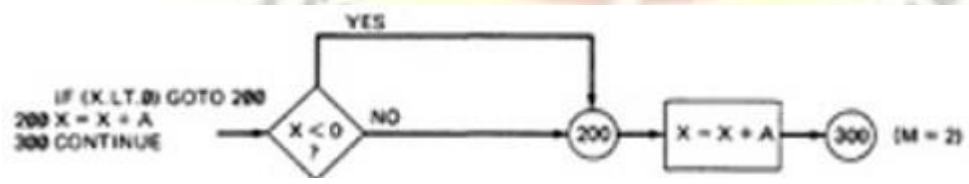
- There are many paths between the entry and exit of a typical routine.
- Every decision doubles the number of potential paths. And every loop multiplies the number of potential paths by the number of different iteration values possible for the loop.

Defining complete testing:

0. Exercise every path from entry to exit
1. Exercise every statement or instruction at least once
2. 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.

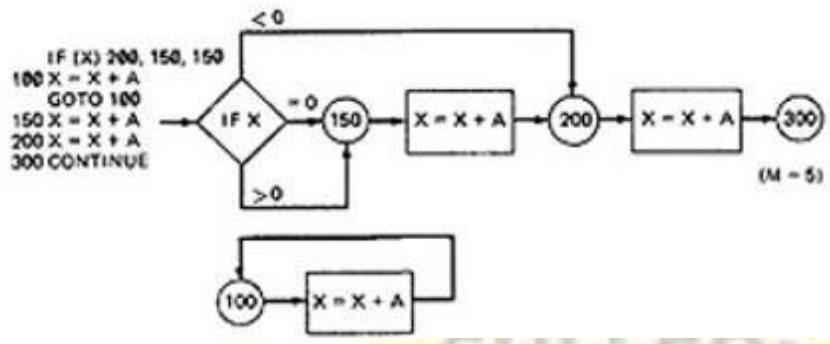


For X negative, the output is X + A, while for X greater than or equal to zero, the output is X + 2A. Following prescription 2 and executing every statement, but not every branch, would not reveal the bug in the following incorrect version:



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.)**

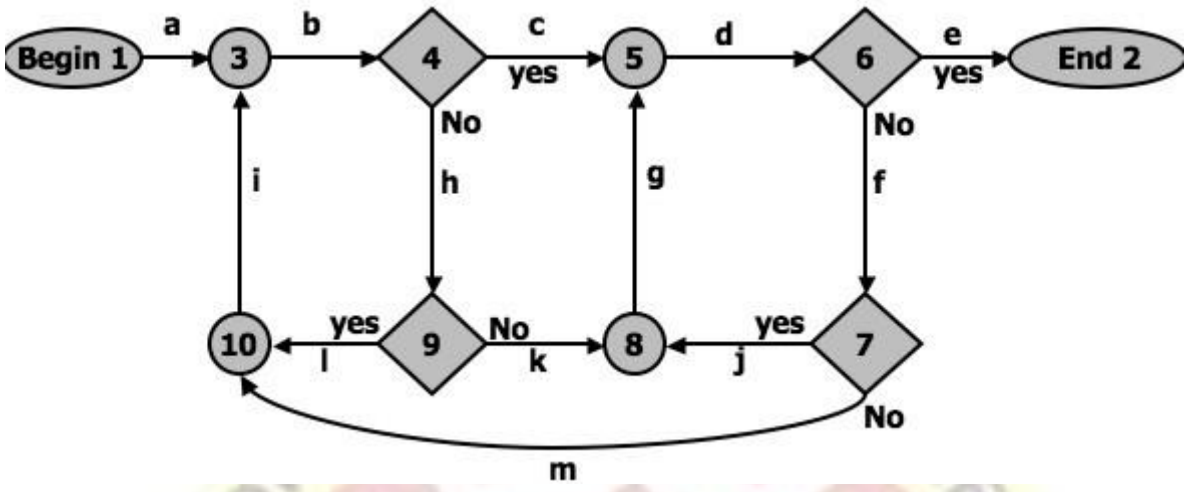
  The high probability paths are always thoroughly tested if only to demonstrate that the system works properly.

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:

An example flowgraph to explain path selection



Practical Suggestions in Path Testing:

0. Draw the control flow graph on a single sheet of paper.
1. Make several copies - as many as you will need for coverage (C1+C2) and several more.
2. Use a yellow highlighting marker to trace paths. Copy the paths onto master sheets.
3. Continue tracing paths until all lines on the master sheet are covered, indicating that you appear to have achieved C1+C2.

4. As you trace the paths, create a table that shows the paths, the coverage status of each process, and each decision.
5. The above paths lead to the following table considering

After you have traced a covering path set on the master sheet and filled in thetable for

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		✓	✓	✓	✓	✓	✓			✓				✓

every path, check the following:

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

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

0. 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.
1. Could the loop-control variable be negative? Could it appear to specify a negative number of iterations? What happens to such a value?
2. One pass through the loop.
3. Two passes through the loop.
4. A typical number of iterations, unless covered by a previous test.
5. One less than the maximum number of iterations.
6. The maximum number of iterations.
7. 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

8. 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?
9. The minimum number of iterations.
10. One more than the minimum number of iterations.
11. Once, unless covered by a previous test.
12. Twice, unless covered by a previous test.
13. A typical value.
14. One less than the maximum value.
15. The maximum number of iterations.
16. Attempt one more than the maximum number of iterations.

CASE 3: Single loops with excluded values

- Treat single loops with excluded values as two sets of tests consisting of loops without excluded values, such as case 1 and 2 above.
- Example, the total range of the loop control variable was 1 to 20, but that values 7,8,9,10 were excluded. The two sets of tests are 1-6 and 11-20.
- The test cases to attempt would be 0,1,2,4,6,7 for the first range and 10,11,15,19,20,21 for the second range.

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

Nested Loops:

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

1. Start at the inner most loop. Set all the outer loops to their minimum values.
2. Test the minimum, minimum+1, typical, maximum-1 , and maximum for the innermost loop, while holding the outer loops at their minimum iteration parameter values. Expand the tests as required for out of range and excluded values.
3. If you've done the outmost loop, GOTO step 5, else move out one loop and set it up as in step

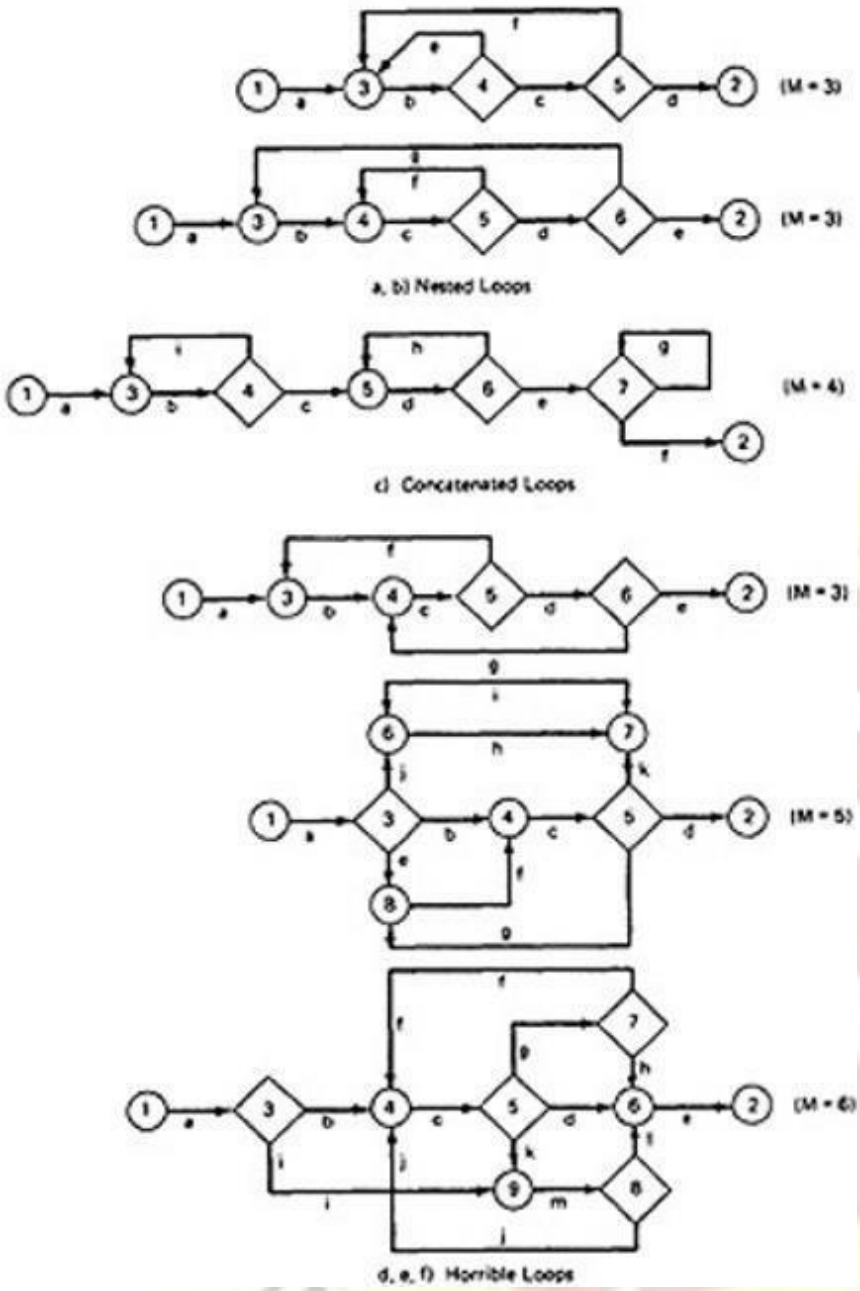
2 with all other loops set to typical values.
4. Continue outward in this manner until all loops have been covered.
5. Do all the cases for all loops in the nest simultaneously.

Concatenated Loops:

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

Horrible Loops:

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



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, ...
A: Z := 7 @ GOTO HEM
B: Z := -7 @ GOTO HEM
C: Z := 0 @ GOTO HEM
.....
HEM: DO SOMETHING
.....
HEN: IF Y + Z > 0 GOTO ELL ELSE GOTO EMM
```

The predicate interpretation at HEN depends on the path we took through the first multiway branch. It yields for the three cases respectively, if Y+7>0, Y-7>0, Y>0.
□□The path predicates are the specific form of the predicates of the decisions along the selected path after interpretation.

INDEPENDENCE OF VARIABLES AND PREDICATES:

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

CORRELATION OF VARIABLES AND PREDICATES:

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

PATH PREDICATES 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.
- Some times a predicate can have an OR in it.
- Example:

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

Boolean algebra notation to denote the boolean expression:

ABCD+EBCD=(A+E)BCD

PREDICATE COVERAGE:

- **Compound Predicate:** Predicates of the form A OR B, A AND B and more complicated boolean expressions are called as compound predicates.
- Some times even a simple predicate becomes compound after interpretation. Example: the predicate if (x=17) whose opposite branch is if x.NE.17 which is equivalent to x>17 . Or. X<17.
- Predicate coverage is being the achieving of all possible combinations of truth values corresponding to the selected path have been explored under some test.
- As achieving the desired direction at a given decision could still hide bugs in the associated predicates.

TESTING BLINDNESS:

- Testing Blindness is a pathological (harmful) situation in which the desired path is achieved for the wrong reason.
- There are three types of Testing Blindness:

1. Assignment Blindness:

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

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

2. Equality Blindness:

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

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

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

3. Self Blindness:

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

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

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

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 flowgraph 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+BCEFGHIJ KL

- 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 cant find a solution to any of the sets of inequalities, the path is un achievable.
- The act of finding a set of solutions to the path predicate expression is called **PATH SENSITIZATION**.

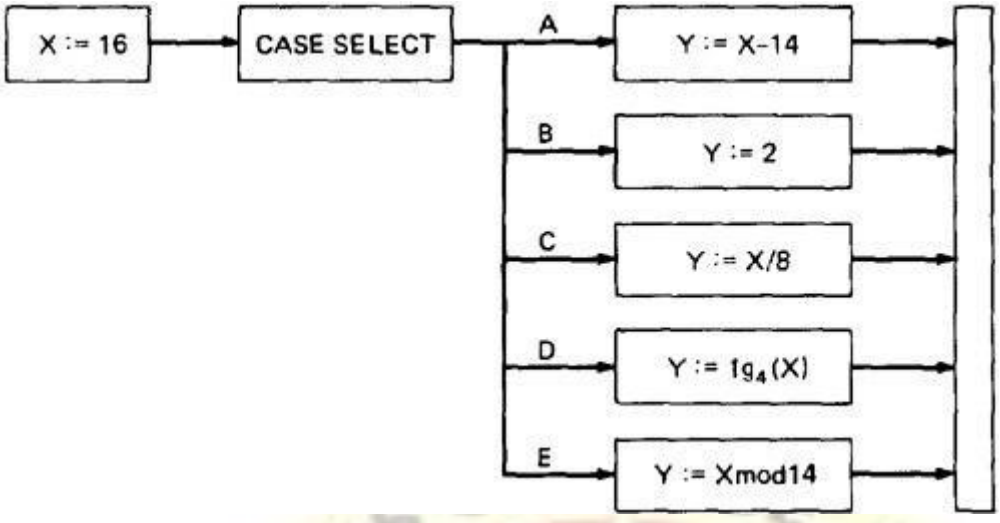
HEURISTIC PROCEDURES FOR SENSITIZING PATHS:

- 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.



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:

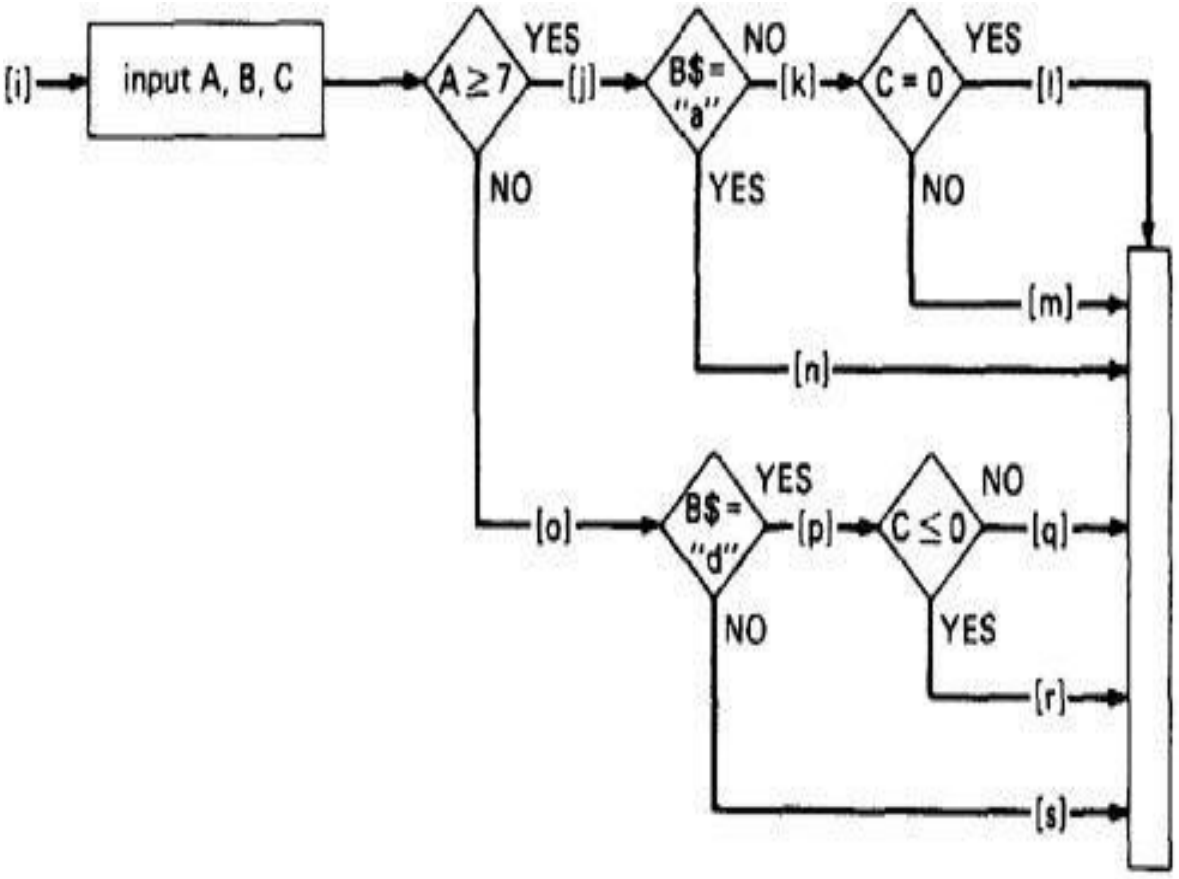
1. Interpretive Trace Program:

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

2. Traversal Marker or Link Marker:

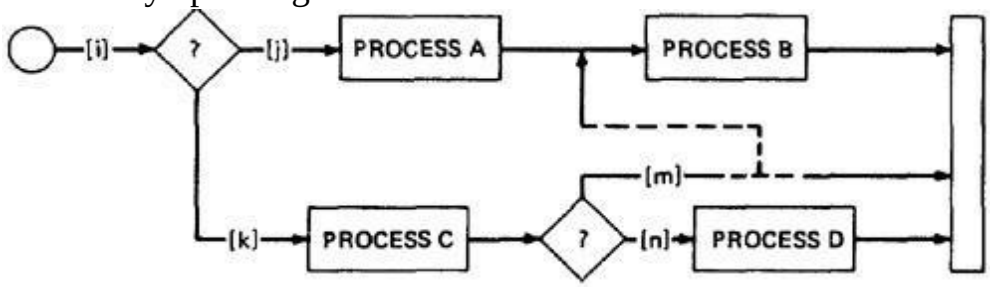
- A simple and effective form of instrumentation is called a traversal marker or link marker.
- Name every link by a lower case letter.

- Instrument the links so that the link's name is recorded when the link is executed.
- The succession of letters produced in going from the routine's entry to its exit should, if there are no bugs, exactly correspond to the path name.



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

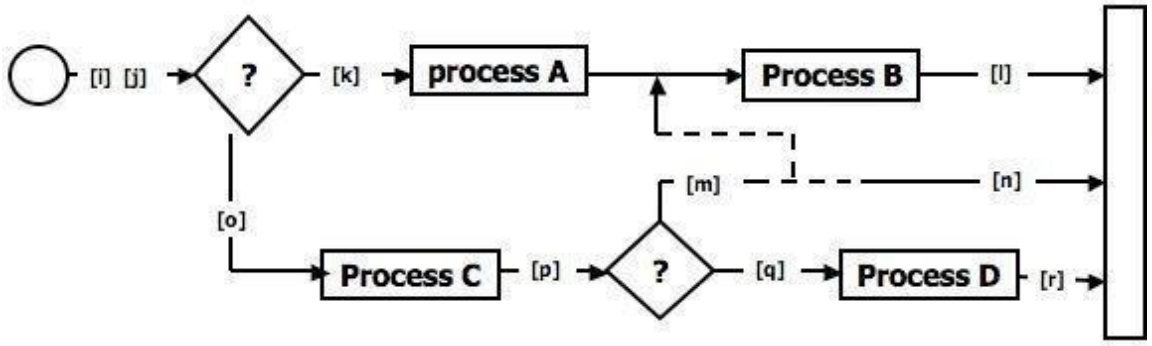


We intended to traverse the ikm path, but because of a rampaging GOTO in the middle of the m link, we go to process B. If coincidental correctness is against us, the outcomes will be the same and we won't know about the bug.

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.

<u>Double Link Marker Instrumentation.</u>



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.

.TRANSACTION FLOWS:

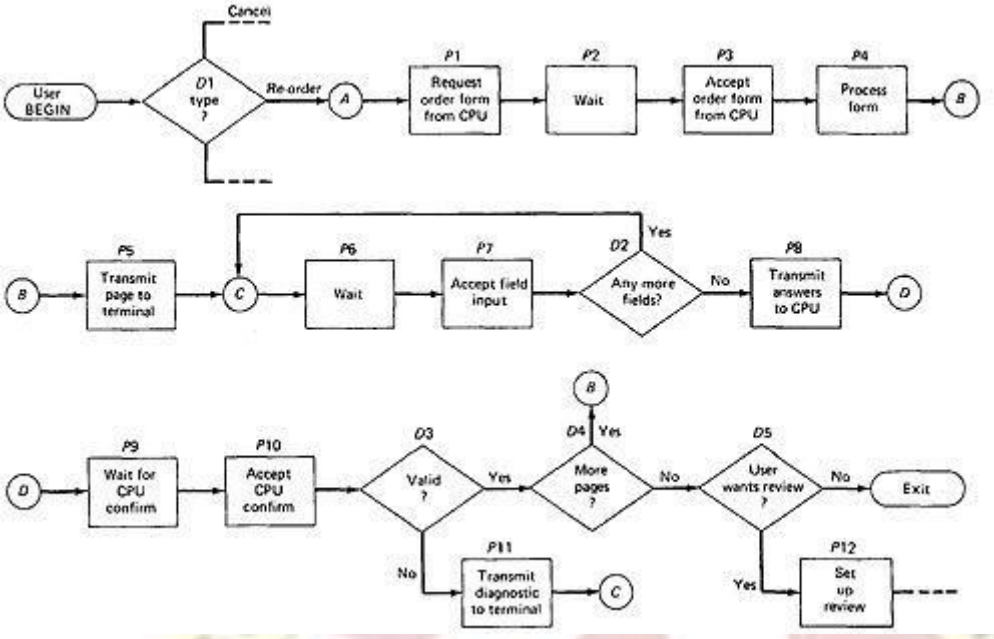
INTRODUCTION:

- 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 begin 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:



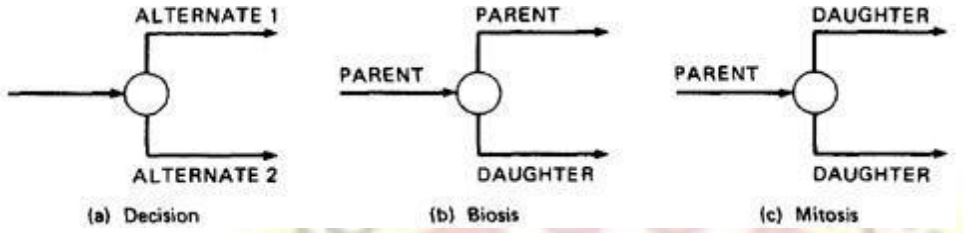
USAGE:

- 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.

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



Mergers: Transaction flow junction points are potentially as troublesome as transaction flow splits. There are three types of junctions: (1) Ordinary Junction (2) Absorption (3) Conjugation
0. **Ordinary Junction:** An ordinary junction which is similar to the junction in a control flow graph. A transaction can arrive either on one link or the other.
1. **Absorption:** In absorption case, the predator transaction absorbs prey transaction. The prey gone but the predator retains its identity
2. **Conjugation:** In conjugation case, the two parent transactions merge to form a new daughter. In keeping with the biological flavor this case is called as conjugation

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

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.

Unit-3

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 been initialized 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 machnes (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:

1. Fetch instruction from memory
2. Interpret instruction
3. Fetch operands
4. Process or Execute
5. Store result
6. Increment program counter
7. GOTO 1

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

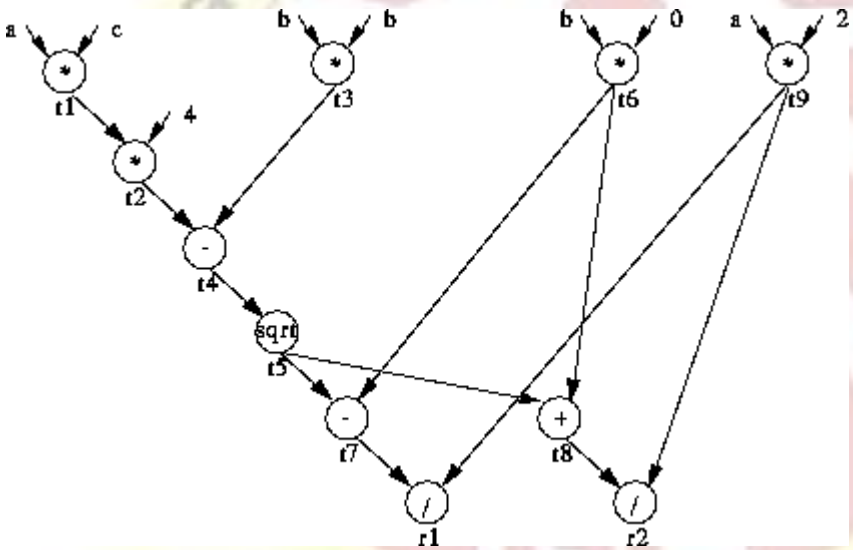
- These machines can fetch several instructions and objects in parallel.
- They can also do arithmetic and logical operations simultaneously on different data objects.
- The decision of how to sequence them depends on the compiler.

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 flowgraphs as such. Rather, we'll use an ordinary control flowgraph 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 an 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.

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:

1. **Defined:** d - defined, created, initialized etc
2. **Killed or undefined:** k - killed, undefined, released etc
3. **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.

2. Killed or Undefined (k):

An object is killed on undefined when it is released or otherwise made unavailable.

3. Usage (u):

- When its contents are no longer known with certitude (with aboslute certainity / 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

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

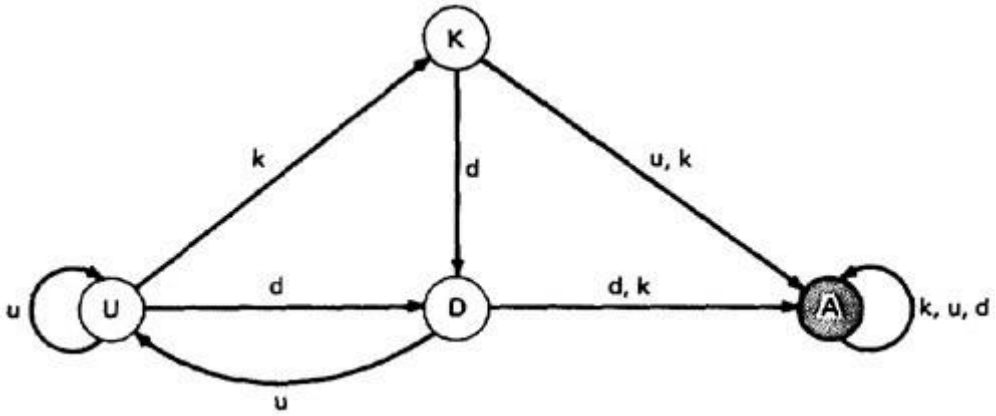
1. **dk** :- probably a bug. Why define the object without using it?
2. **du** :- the normal case. The object is defined and then used.
3. **kd** :- normal situation. An object is killed and then redefined.
4. **kk** :- harmless but probably buggy. Did you want to be sure it was really killed?
5. **ku** :- a bug. the object doesnot exist.
6. **ud** :- usually not a bug because the language permits reassignment at almost any time.
7. **uk** :- normal situation.
8. **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:
0. **-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.
1. **-d** :- okay. This is just the first definition along this path.
2. **-u** :- possibly anomalous. Not anomalous if the variable is global and has been previously defined.
3. **k-** :- not anomalous. The last thing done on this path was to kill the variable.
4. **d-** :- possibly anomalous. The variable was defined and not used on this path. But this could be a global definition.
5. **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:
0. **K** :- undefined, previously killed, doesnot exist
1. **D** :- defined but not yet used for anything
2. **U** :- has been used for computation or in predicate
3. **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.



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 doesnot 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 exists 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 flowgraph..

Here we annotate each link with symbols (for example, d, k, u, c, 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:

0. To every statement there is a node, whose name is unique. Every node has at least one out link and at least one in link except for exit nodes and entry nodes.
1. 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.
2. 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.
3. Predicate nodes (e.g., IF-THEN-ELSE, DO WHILE, CASE) are weighted with the p - use(s) on every outlink, appropriate to that outlink.
4. 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.
5. 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.
6. 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.

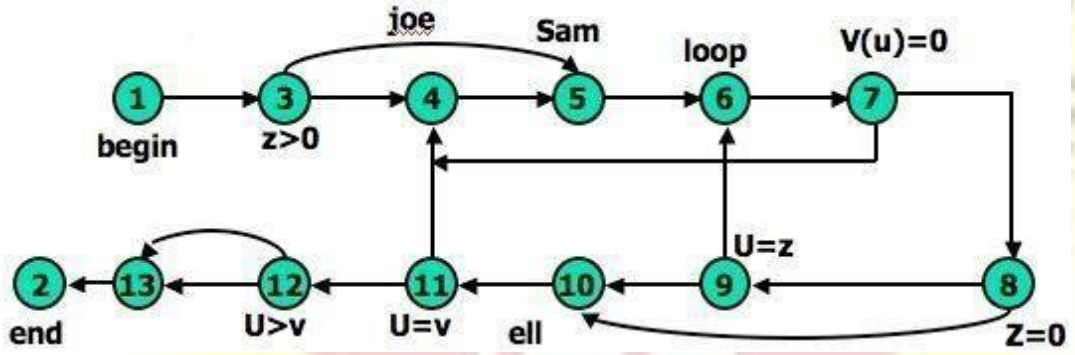
Program Example (PDL)

CODE* (PDL)

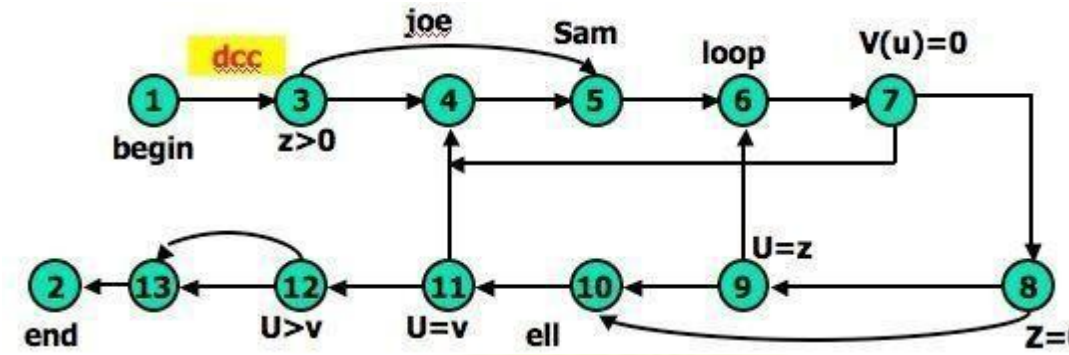
```
        INPUT X, Y                      V(U−1):=V(U+1) + U(V−1)
        Z := X + Y                  ELL:V(U+U(V)) := U + V
        V := X − Y                      IF U = V GOTO JOE
        IF Z >=Ø GOTO SAM               IF U > V THEN U := Z
  JOE:  Z := Z − 1                      Z := U
  SAM: 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

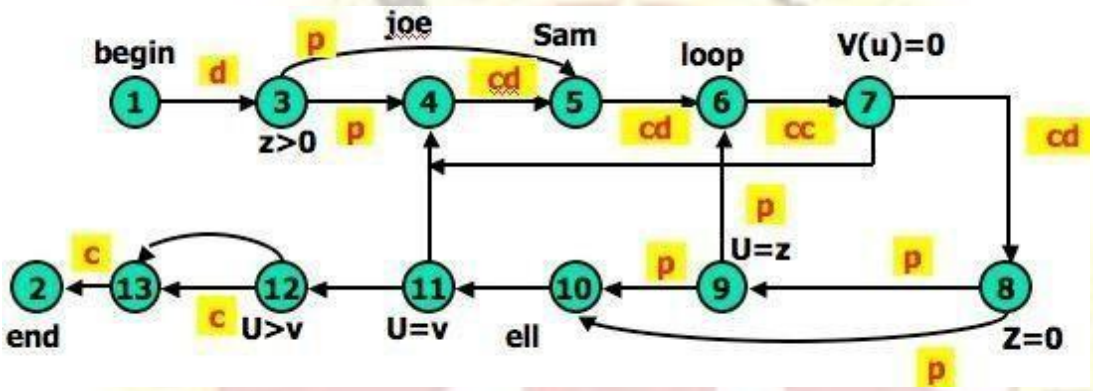
Unannotated flowgraph



Control flowgraph annotated for X and Y data flows.



Control flowgraph annotated for Z 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 subpaths 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:

1. **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).

2. **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.

3. **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.

4. A **du path** from node i to k is a path segment such that if the last link has a computational use of X, then the path is simple and definition-clear; if the penultimate (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 flowgraph. 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:

0. **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 use of that definition be exercised under 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.

1. **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.

2. **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 in this 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.

3. **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.
4.
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.

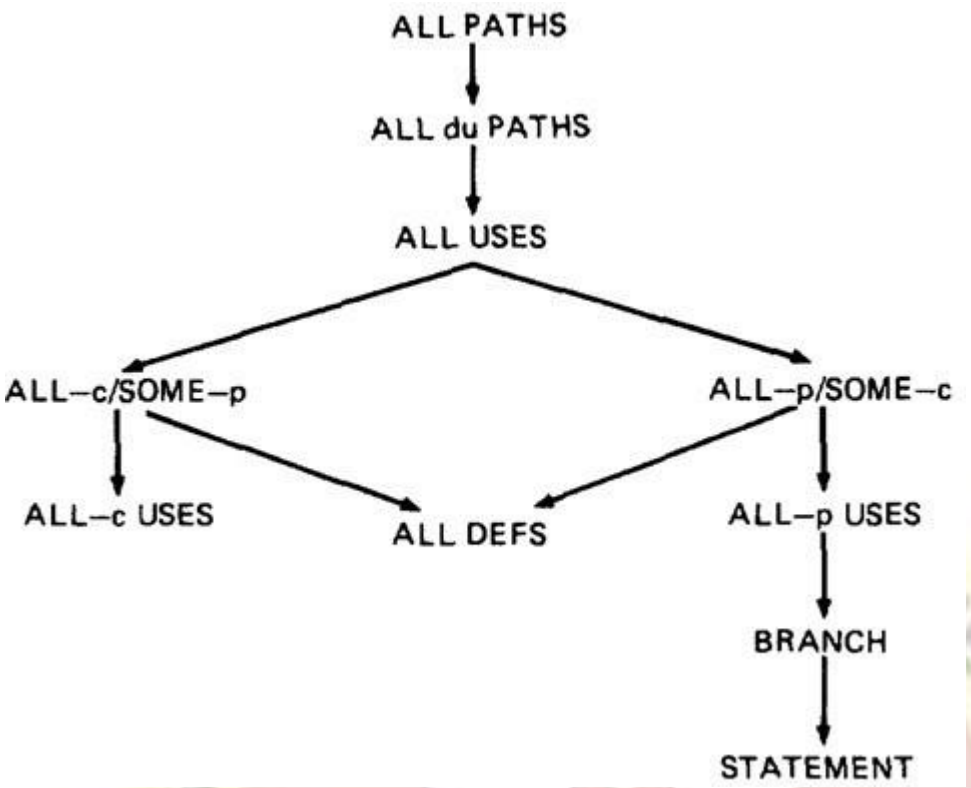
5. **All Definitions Strategy (AD) :** The all definitions strategy asks only every definition of every variable be covered by atleast one use of that variable, be that use a computational use or a predicate use.

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.

6. **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:
The below figure compares 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.



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.

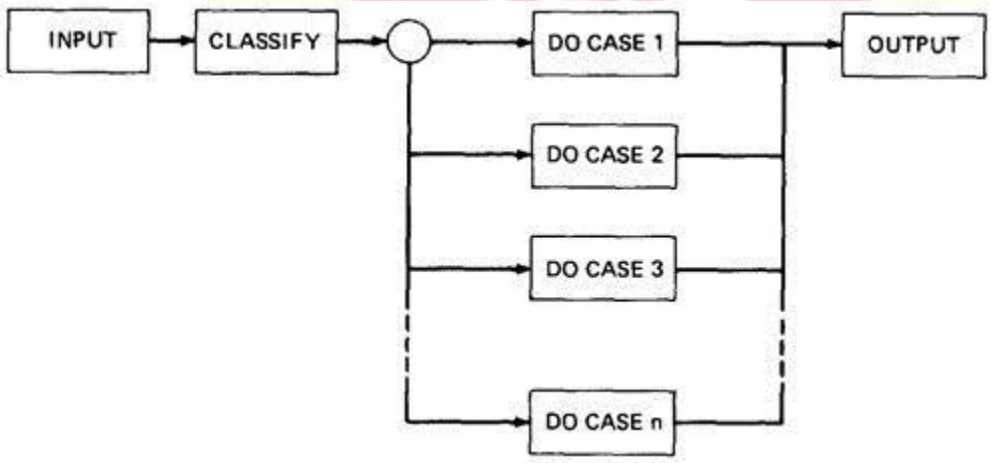
DOMAIN TESTING:

- **INTRODUCTION:**
  - **Domain:** In mathematics, domain is a set of possible values of an independant 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.
  - 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.

Schematic Representation of Domain Testing.



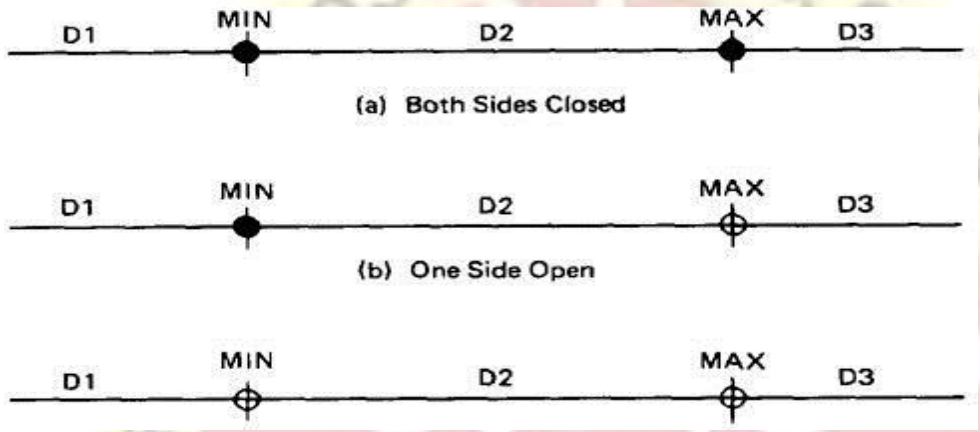
- **A DOMAIN IS A SET:**
  - An input domain is a set.
  - If the source language supports set definitions (E.g. PASCAL set types and C enumerated types) less testing is needed because the compiler does much of it for us.
  - Domain testing does not work well with arbitrary discrete sets of data objects.
  - 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 flowgraph.
  - Conversely, if domain testing is applied to specifications, interpretation is based on a specified data flowgraph 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).
  - 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.



- **DOMAIN DIMENSIONALITY:**
  - Every input variable adds one dimension to the domain.
  - One variable defines domains on a number line.
  - Two variables define planar domains.
  - 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.

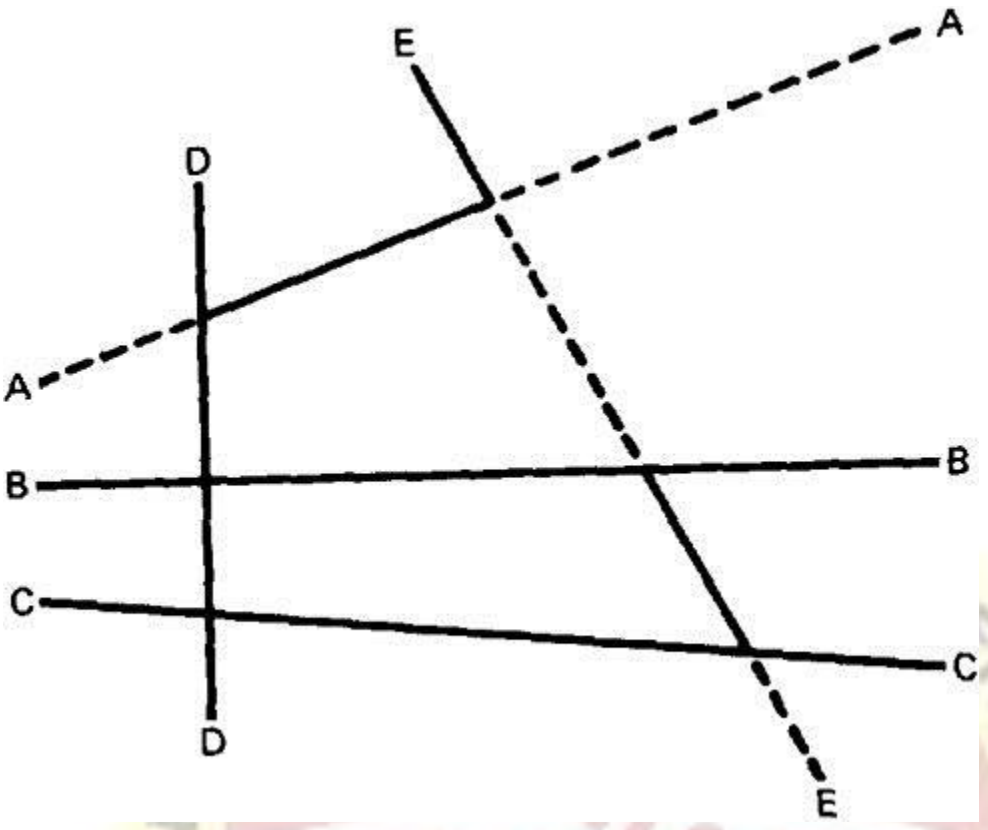
- Many different bugs can result in domain errors. Some of them are:

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 it self or multiplied by zero. So the floating point zero check to be done against a 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.
- **Overspecified Domains:**he 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 a domain. 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.

- **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 points to determine a 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 shows some incomplete boundaries. Boundaries A and E have gaps.
  - 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.

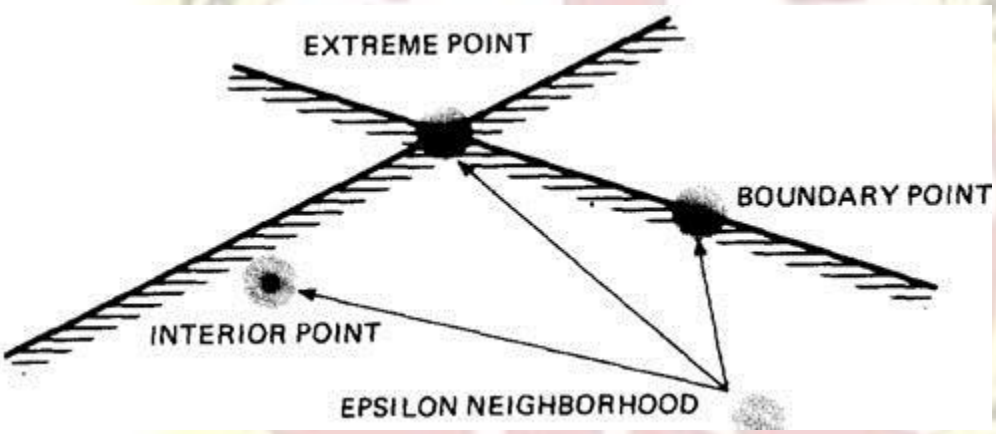


DOMAIN TESTING:

- **DOMAIN TESTING STRATEGY:** The domain-testing strategy is simple, although possibly tedious (slow).
    1. Domains are defined by their boundaries; therefore, domain testing concentrates test points on or near boundaries.
    2. Classify what can go wrong with boundaries, then define a test strategy for each case. Pick enough points to test for all recognized kinds of boundary errors.
    3. Because every boundary serves at least two different domains, test points used to check one domain can also be used to check adjacent domains. Remove redundant test points.
    4. Run the tests and by posttest analysis (the tedious part) determine if any boundaries are faulty and if so, how.
    5. Run enough tests to verify every boundary of every domain.

- **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.

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.

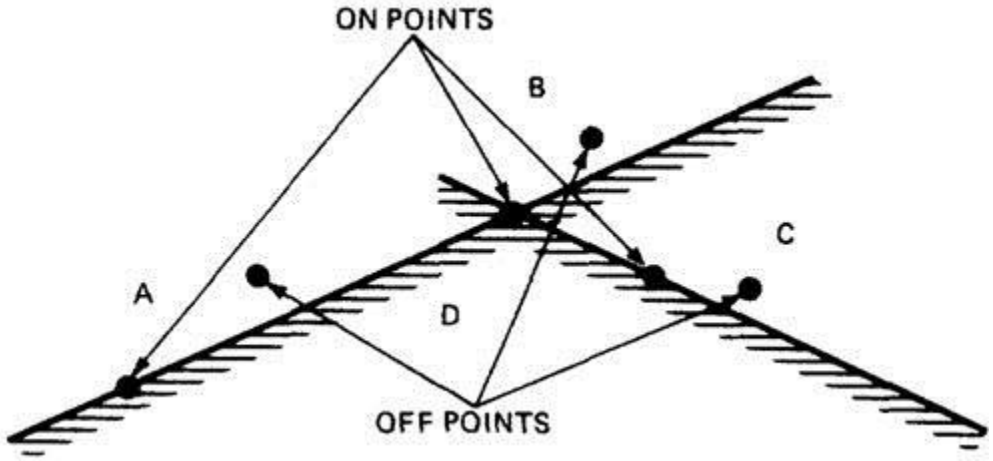
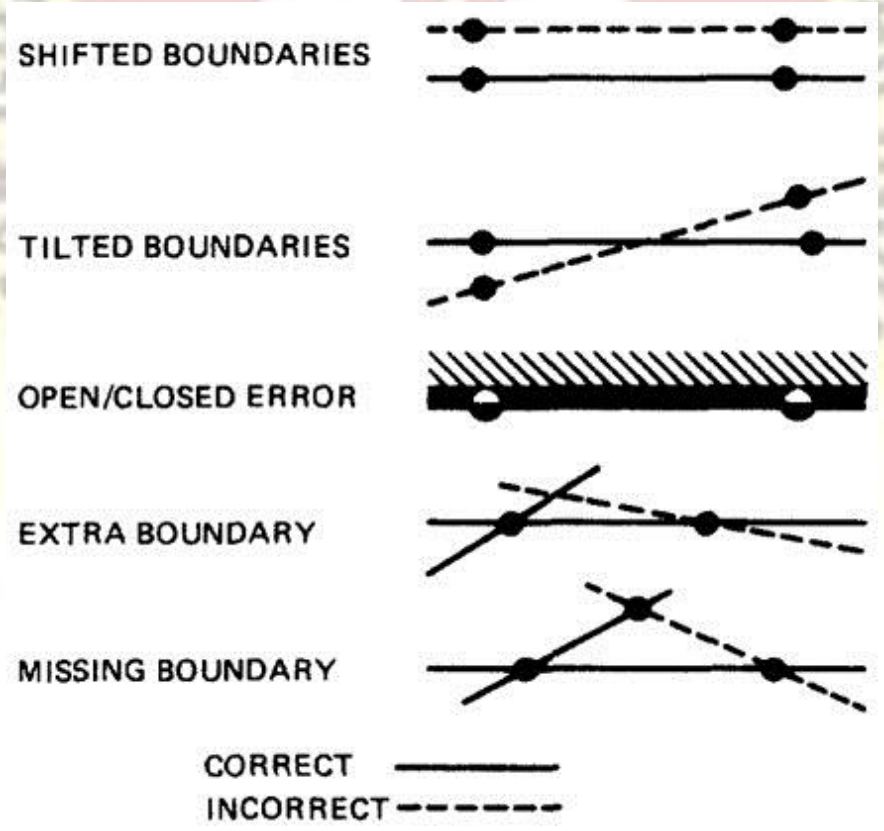


Figure shows generic domain bugs: closure bug, shifted boundaries, tilted boundaries, extra boundary, missing boundary.



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 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:

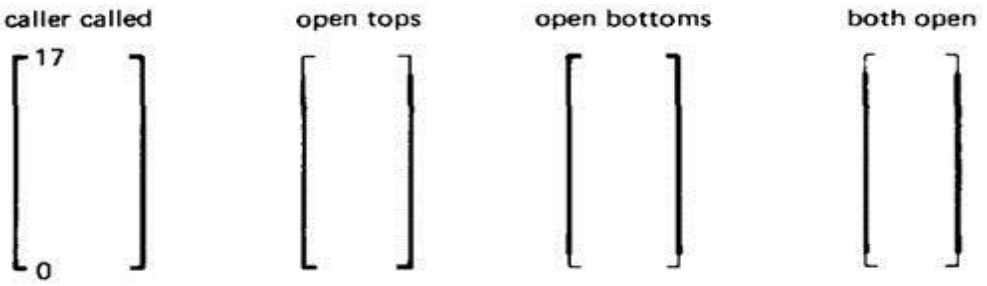
- Recall that we defined integration testing as testing the correctness of the interface between two otherwise correct components.
- Components A and B have been demonstrated to satisfy their component tests, and as part of the act of integrating them we want to investigate possible inconsistencies across 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).

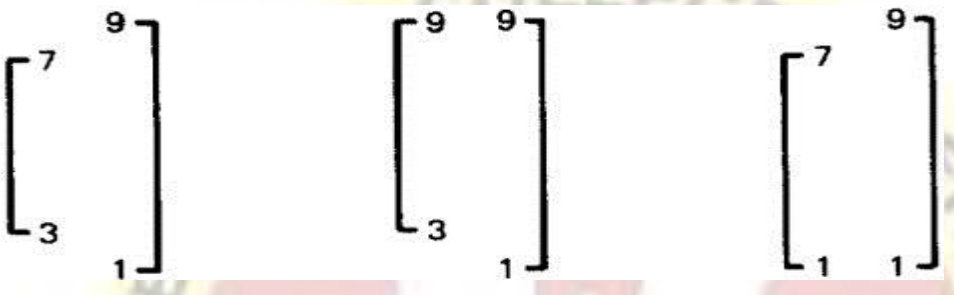
- **DOMAINS AND RANGE:**
  - The set of output values produced by a function is called the **range** of the function, in contrast with the **domain**, which is the set of input values over which the function is defined.
  - For most testing, our aim has been to specify input values and to predict and/or 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 span 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 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.



SPAN COMPATIBILITY:



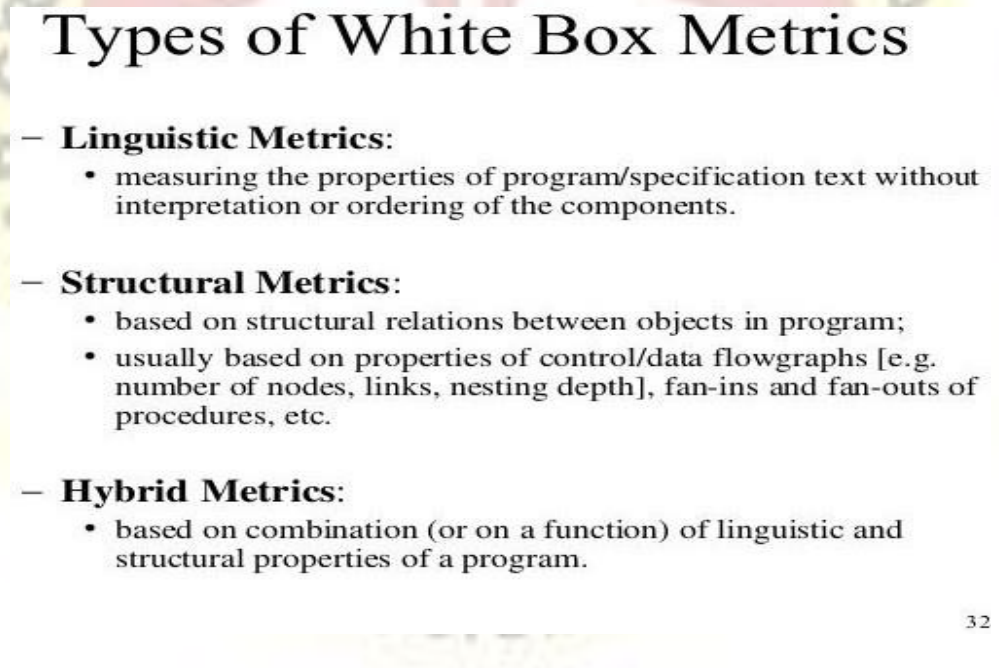
- **INTERFACE RANGE / DOMAIN COMPATIBILITY TESTING:**
  - For interface testing, bugs are more likely to concern single variables rather than peculiar combinations of two or more variables.
  - Test every input variable independently of other input variables to confirm compatibility of the caller's range 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-4

Software Testing Metrics

Software Testing Metrics are the quantitative measures used to estimate the progress, quality, productivity and health of the software testing process. The goal of software testing metrics is to improve the efficiency and effectiveness in the software testing process and to help make better decisions for further testing process by providing reliable data about the testing process.

A Metric defines in quantitative terms the degree to which a system, system component, or process possesses a given attribute. The ideal example to understand metrics would be a weekly mileage of a car compared to its ideal mileage recommended by the manufacturer.



Linguistic Metrics: Based on measuring properties of program text without interpreting what the text means. – E.g., LOC.

Structural Metrics: Based on structural relations between the objects in a program. – E.g., number of nodes and links in a control flowgraph.

Lines of code (LOC)

• LOC is used as a measure of software complexity.

• This metric is just as good as source listing weight if we assume consistency w.r.t. paper and font size. • Makes as much sense (or nonsense) to say: – ‑This is a 2 pound program‖

• as it is to say: – ‑This is a 100,000 line program.‖

LOC

- Various studies indicate:
  - error rates ranging from 0.04% to 7% when measured against statement counts;
  - LOC is as good as other metrics for small programs
  - LOC is optimistic for bigger programs.
  - LOC appears to be rather linear for small programs (<100 lines),
  - but increases non-linearity with program size.
  - Correlates well with maintenance costs
  - Usually better than simple guesses or nothing at all.

35

Structural Metrics

- McCabe's Cyclomatic Complexity
  - Control Flow Graphs
- Information Flow Metric

44

McCabe Cyclomatic Complexity

McCabe's cyclomatic complexity is a software quality metric that quantifies the complexity of a software program. Complexity is inferred by measuring the number of linearly independent paths through the program. The higher the number the more complex the code.

The Significance of the McCabe Number

Measurement of McCabe's cyclomatic complexity metric ensures that developers are sensitive to the fact that programs with high McCabe numbers (e.g. > 10) are likely to be difficult to understand and therefore have a higher probability of containing defects. The cyclomatic complexity number also indicates the number of test cases that would have to be written to execute all paths in a program.

Calculating the McCabe Number

Cyclomatic complexity is derived from the control flow graph of a program as follows:

Cyclomatic complexity (CC) = E - N + 2P
Where:
P = number of disconnected parts of the flow graph (e.g. a calling program and a subroutine)
E = number of edges (transfers of control)
N = number of nodes (sequential group of statements containing only one transfer of control)
McCabe Cyclomatic Complexity
(Alias: McCabe number)

> *Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.*
> *- Alan Perlis, American Scientist*

McCabe's cyclomatic complexity is a software quality metric that quantifies the complexity of a software program. Complexity is inferred by measuring the number of linearly independent paths through the program. The higher the number the more complex the code.

The Significance of the McCabe Number

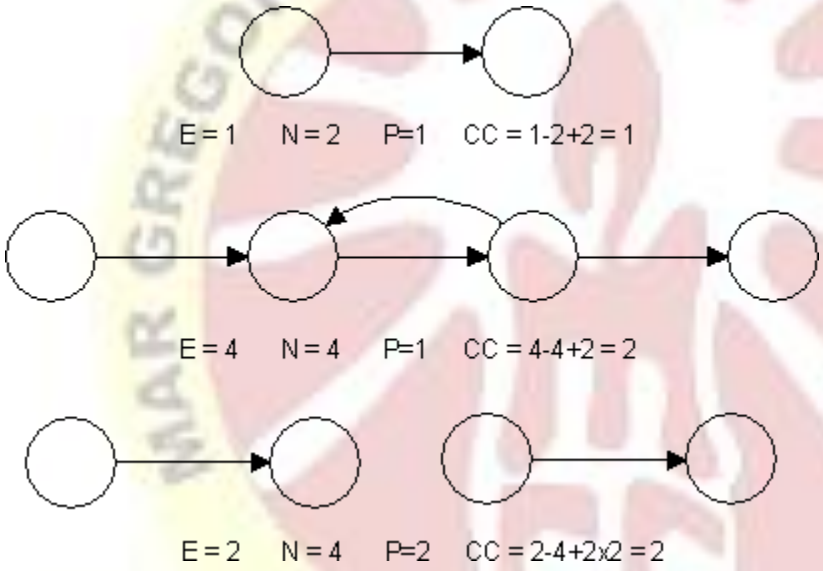
Measurement of McCabe's cyclomatic complexity metric ensures that developers are sensitive to the fact that programs with high McCabe numbers (e.g. > 10) are likely to be difficult to understand and therefore have a higher probability of containing defects. The cyclomatic complexity number also indicates the number of test cases that would have to be written to execute all paths in a program.

Calculating the McCabe Number

Cyclomatic complexity is derived from the control flow graph of a program as follows:

Cyclomatic complexity (CC) = E - N + 2P
Where:
P = number of disconnected parts of the flow graph (e.g. a calling program and a subroutine)
E = number of edges (transfers of control)
N = number of nodes (sequential group of statements containing only one transfer of control)

Examples of McCabe Number Calculations



Halstead's Software Metrics

According to Halstead's "A computer program is an implementation of an algorithm considered to be a collection of tokens which can be classified as either operators or operand."

The basic measures are

n1 = count of unique operators.
n2 = count of unique operands.
N1 = count of total occurrences of operators.
N2 = count of total occurrence of operands.

In terms of the total tokens used, the size of the program can be expressed as N = N1 + N2.

Estimated Program Length

According to Halstead, The first Hypothesis of software science is that the length of a well-structured program is a function only of the number of unique operators and operands.

$N=N1+N2$

And estimated program length is denoted by N^

$N^ = n1\log_2 n1 + n2\log_2 n2$

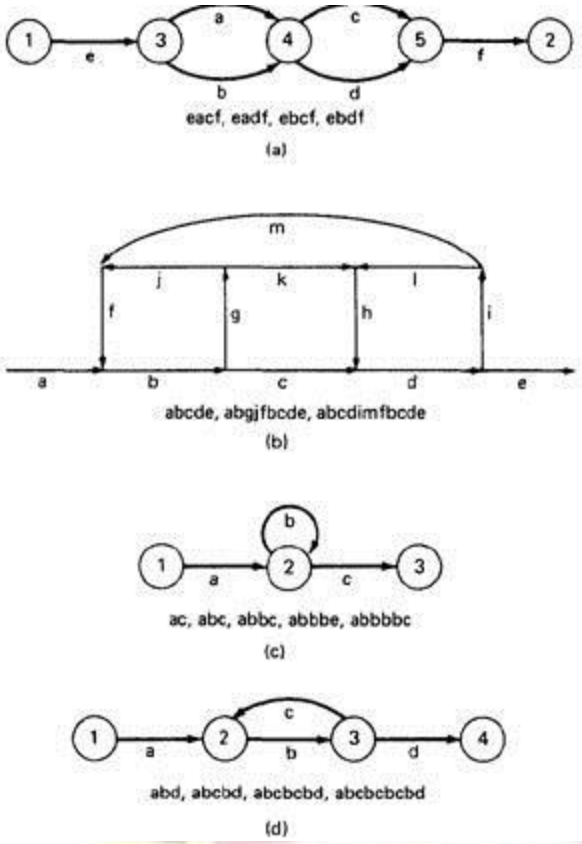
The following alternate expressions have been published to estimate program length:

- $N_J = \log_2 (n1!) + \log_2 (n2!)$
- $N_B = n1 * \log_2 n2 + n2 * \log 2 n1$
- $N_C = n1 * sqrt(n1) + n2 * sqrt(n2)$
- $N_S = (n * \log_2 n) / 2$

PATH PRODUCTS AND PATH EXPRESSION:

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.**



- **PATH EXPRESSION:**
  - Consider a pair of nodes in a graph and the set of paths between those node.
  - 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
- **RULE 1:** A(BC)=(AB)C=ABC

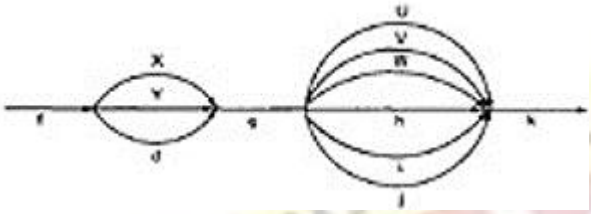
  where A,B,C are path names, set of path names or path expressions.

- 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.



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)

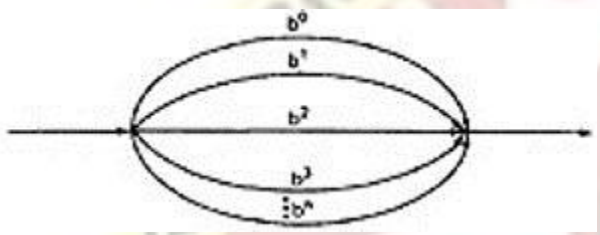
- If a set consists of paths names and a member of that set is added to it, the "new" name, which is already in that set of names, contributes nothing and can be ignored.

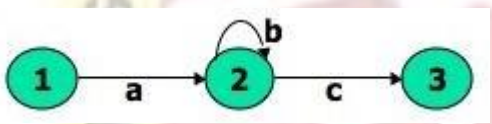
- For example,
- if X=a+aa+abc+abcd+def then
  X+a = X+aa = X+abc = X+abcd = X+def = X

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+.....





The path expression for the above figure is denoted by the notation:

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

Syntax Testing

- System inputs must be validated. Internal and external inputs conform to formats: – Textual format of data input from users. – File formats. – Database schemata.
- Data formats can be mechanically converted into many input data validation tests.
- Such a conversion is easy when the input is expressed in a formal notation such as BNF (Backus-Naur Form).

Syntax Testing Steps

- Identify the target language or format.
- Define the syntax of the language, formally, in a notation such as BNF.
- Test and Debug the syntax: – Test the ‒normal‖ conditions by covering the BNF syntax graph of the input language. (minimum requirement) – Test the ‒garbage‖ conditions by testing the system against invalid data. (high payoff)

How to Find the Syntax

- Every input has a syntax.
- The syntax may be:
  – formally specified
  – undocumented
  – just understood
- … but it does exist!
- Testers need a formal specification to test the syntax and create useful ‒garbage‖.

BNF

Syntax is defined in BNF as a set of definitions. Each definition may in-turn refer to other definitions or to itself.

The LHS of a definition is the name given to the collection of objects on the RHS.

– ::= means ‒is defined as‖.

– | means ‒or‖.

– * means ‒zero or more occurrences‖.

– + means ‒one or more occurrences‖.

Test Case Generation

There are three possible kinds of incorrect actions:

– Recognizer does not recognize a good string.

– Recognizer accepts a bad string.

– Recognizer crashes during attempt to recognize a string.

• Even small BNF specifications lead to many good strings and far more bad strings.

• There is neither time nor need to test all strings.

Testing Strategy

- Create one error at a time, while keeping all other components of the input string correct.
- Once a complete set of tests has been specified for single errors, do the same for double errors, then triple, errors, ...
- Focus on one level at a time and keep the level above and below as correct as you can.

Delimiter Errors

Delimiters are characters or strings placed between two fields to denote where one ends and the other begin.

• Delimiter Problems:

– Missing delimiter. e.g., (x+y

– Wrong delimiter. e.g., (x+y]

– Not a delimiter. e.g., (x+y 1

– Poorly matched delimiters. e.g., (x+y))

Sources of Syntax

- Designer-Tester Cooperation
- Manuals
- Help Screens
- Design Documents
- Prototypes
- Programmer Interviews
- Experimental (hacking)

Unit-5 :

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.
  - 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.
  - Higher-order logic systems are needed and used for formal specifications.
  - Much of logical analysis can be and is embedded in tools. But these tools incorporate methods to simplify, transform, and

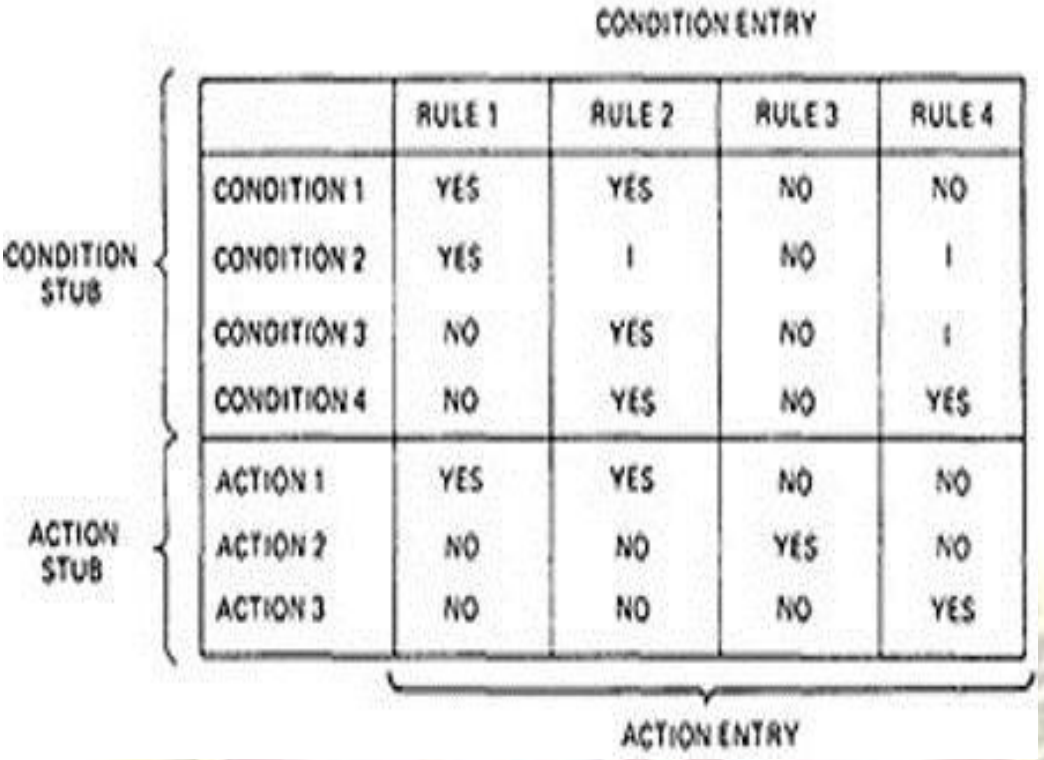
check specifications, and the methods are to a large extent based on boolean algebra.

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:



-
- 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.

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				

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

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.

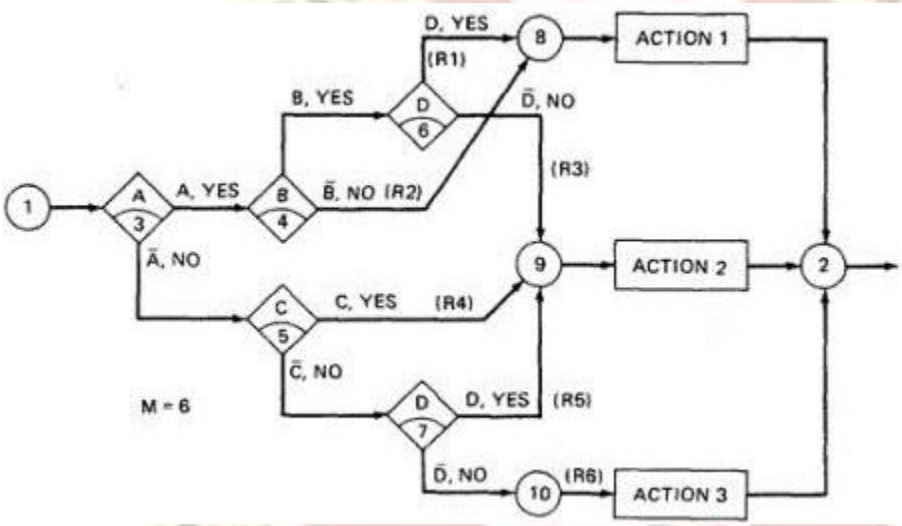
DECISION-TABLES AS BASIS FOR TEST CASE DESIGN:

5. The specification is given as a decision table or can be easily converted into one.
6. 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.
7. 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.

8. Once a rule is satisfied and an action selected, no other rule need be examined.
9. 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.
- Figure 6.4 shows a program segment that consists of a decision tree.
- These decisions, in various combinations, can lead to actions 1, 2, or 3



- 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

	RULE 1	RULE 2	RULE 3	RULE 4	RULE 5	RULE 6

CONDITION A	YES	YES	YES	NO	NO	NO
CONDITION B	YES	NOI	YES	I	I	I
CONDITION C	I	I	I	YES	NO	NO
CONDITION D	YES		NO	I	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

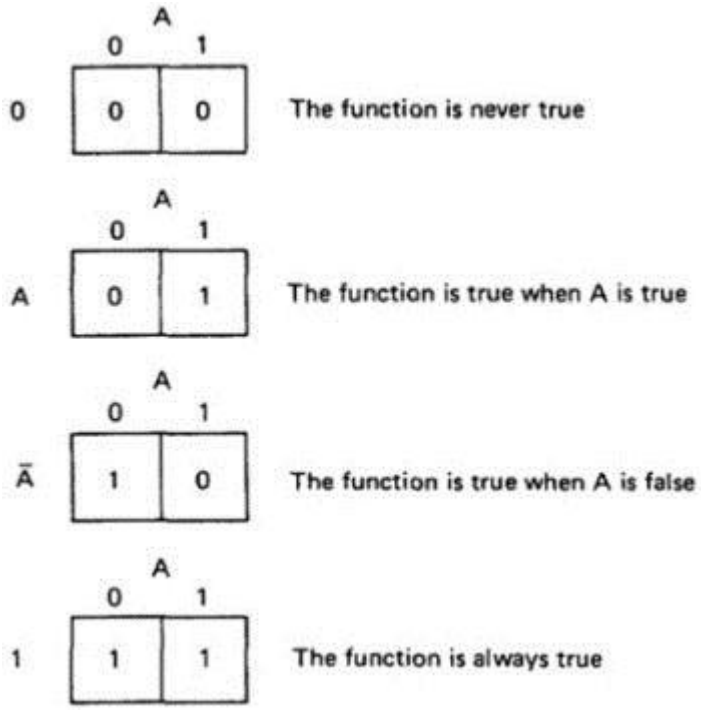
KV CHARTS

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:

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



- The charts show all possible truth values that the variable A can have.
- 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.

STATES, STATE GRAPHS AND TRANSITION TESTING

OBJECTIVE:

To know how state testing strategies are based on the use of finite state machine models for software structure, software behavior, or specifications of software behavior.

Introduction:

The finite state machine is as fundamental to software engineering as Boolean algebra to logic.

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

State Graphs:

OBJECTIVE:

State graph is used to represent states, links, and transitions from one state to involves a program that detects the character sequence –in the graph.

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

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

1. Reverse gear

2. Neutral gear

3. First gear

4. Second gear

5. Third gear

6. Fourth gear State graph –

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

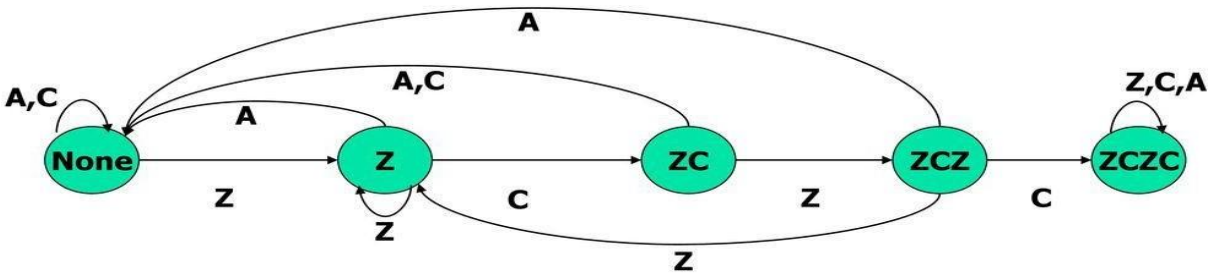
7. Neither ZCZC nor any part of it has been detected.

8. Z has been detected.

9. ZC has been detected.

10. ZCZ has been detected.

11. ZCZC has been detected.



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

States are represented by Nodes.

State are numbered or may identified by words or whatever else is convenient

1. Inputs and Transition:

1. 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.

2. Transition are denoted by links that join the states.

3. The input that causes the transition are marked on the link; That is, the inputs are link weights.

4. There is one outlink from every state for every input.

5. 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……‖.

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

2. Outputs

1. An output can be associated with any link.

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

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

4. Outputs are also link weights.

5. If every input associated with a transition causes the same output, then denoted it as: –input1, input2, input3 ...../output‖

3. State Tables:

1.Big state graphs are cluttered and hard to follow.

2. 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.

3. 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

4. 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.

5. Software Implementation

Implementation and Operation:

1. There are four tables involved:

2. A table or process that encodes the input values into a compactlist (INPUT_CODE_TABLE).

3. A table that specifies the next state for every combination of state and input code (TRANSITION_TABLE).

4. A table or case statement that specifies the output or output code, if any, associated with every state-input combination (OUTPUT_TABLE).

5. A table that stores the present state of every device or process that uses the same state table—e.g., one entry per tape transport (DEVICE_TABLE).

6. The routine operates as follows, where # means concatenation: BEGIN

a) PRESENT_STATE := DEVICE_TABLE(DEVICE_NAME)
b) ACCEPT INPUT_VALUE
c) INPUT_CODE := INPUT_CODE_TABLE(INPUT_VALUE)
d) POINTER := INPUT_CODE#PRESENT STATE
e) NEW_STATE := TRANSITION_TABLE(POINTER)
f) OUTPUT_CODE := OUTPUT_TABLE(POINTER)
g) CALL OUTPUT_HANDLER(OUTPUT_CODE)
h) DEVICE_TABLE(DEVICE_ NAME) := NEW_STATE END

State Codes and State-Symbol Product:

1. The term state-symbol product is used to mean the value obtained by any scheme used to convert the combined state and input code into a pointer to a compact table without holes.

2. ‒state codes‖ in the context of finite-state machines, we mean the (possibly) hypothetical integer used to denote the state and not the actual form of the state code that could result from an encoding process.

Good State graphs and bad State graphs

OBJECTIVE: student should find out state graphs which are reachable and non reachable states according to the given specifications or not. To check how equivalent states are possible with set of inputs and outputs

Here are some principles for judging:

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

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

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

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

Number of states:

1. The number of states in a state graph is the number of states we choose to recognize or model.

2. In practice, the state is directly or indirectly recorded as a combination of values of variables that appear in the data base.

3. Find the number of states as follows:

Impossible States:

- Sometimes some combinations of factors may appear to be impossible.
- 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

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.

Dead States:

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

2. This is not necessarily a bug but it is suspicious.

STATE TESTING

- Impacts of Bugs:
- Wrong number of states.
- Wrong transition for a given state-input combination.
- Wrong output for a given transition.
- Pairs of states or sets of states that are inadvertently made equivalent (factor lost).
- States or sets of states that are split to create inequivalent duplicates.
- States or sets of states that have become dead.
- States or sets of states that have become unreachable.

Principles of State Testing: The starting point of state testing is:
1. Define a set of covering input sequences that get back to the initial state when starting from the initial state.
2. For each step in each input sequence, define the expected next state, the expected transition, and the expected output code. A set of tests, then, consists of three sets of sequences:

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

Limitations and Extensions

1. State transition coverage in a state graph model does not guarantee complete testing.

2. How defines a hierarchy of paths and methods for combining paths to produce covers of state graphs.

3. The simplest is called a ―0 switch‖ which corresponds to testing each transition individually.

4. The next level consists of testing transitions sequences consisting of two transitions called ―1 switches‖.

5. The maximum length switch is ―n1 switch‖ where there are n number of states

6. A set of input sequences that provide coverage of all nodes and links is a mandatory minimum requirement.

7. In executing state tests, it is essential that means be provided (e.g., instrumentation software) to record the sequence of states (e.g., transitions) resulting from the input sequence and not just the outputs that result from the input sequence.

*****