CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 1
Unit - 1
Year and Semester: IV yr & I Sem
Subject**: Software Testing Methodologies**
Branch: **CSE**
Faculty: **Revathi Lavanya Baggam,** Assistant Professor (CSE)

Purpose of Testing - CO1

- Testing consumes atleast 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 comsumer 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 componenets, and the cost of quality assurance and testing.
  - There is a trade off between quality assurance costs and manufacturing costs: If sufficient time is not spent in quality assurance, the reject rate will be high and so will be the net cost. If inspection is good and all errors are caught as they occur, inspection costs will dominate, and again the net cost will suffer.
  - Testing and Quality assurance costs for 'manufactured' items can be as low as 2% in consumer products or as high as 80% in products such as space-ships, nuclear reactors, and aircrafts, where failures threaten life. Where as the manufacturing cost of a software is trivial.
  - The biggest part of software cost is the cost of bugs: the cost of detecting them, the cost of correcting them, the cost of designing tests that discover them, and the cost of running those tests.
  - For software, quality and productivity are indistinguishable because the cost of a software copy is trivial.
- Testing and Test Design are parts of quality assurance should also focus on bug prevention. *A prevented bug is better than a detected and corrected bug.*
- Phases in a tester's mental life can be categorised into the following 5 phases:
  - **Phase 0: (Until 1956: Debugging Oriented)** There is no difference between testing and debugging. Phase 0 thinking was the norm in early days of software development till testing emerged as a discipline.
  - **Phase 1: (1957-1978: Demonstration Oriented)** The purpose of testing here is to show that software works. Highlighted during the late 1970s. This failed because the probability of showing that software works 'decreases' as testing increases. *i.e.* The more you test, the more likely you'ill 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.
2. **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.
3. *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.

Dichotomies - CO 1

- **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.

-
- **Function Versus Structure:** Tests can be designed from a functional or a structural point of view. In **functional testing**, the program or system is treated as a blackbox. It is subjected to inputs, and its outputs are verified for conformance to specified behaviour. Functional testing takes the user point of view- bother about functionality and features and not the program's implementation. **Structural testing** does look at the implementation details. Things such as programming style, control method, source language, database design, and coding details dominate structural testing.
- Both Structural and functional tests are useful, both have limitations, and both target different kinds of bugs. Functional tets can detect all bugs but would take infinite time to do so. Structural tests are inherently finite but cannot detect all errors even if completely executed.
- **Designer Versus Tester:** Test designer is the person who designs the tests where as the tester is the one actually tests the code. During functional testing, the designer and tester are probably different persons. During unit testing, the tester and the programmer merge into one person.
- Tests designed and executed by the software designers are by nature biased towards structural consideration and therefore suffer the limitations of structural testing.
- **Modularity Versus Efficiency:** A module is a discrete, well-defined, small component of a system. Smaller the modules, difficult to integrate; larger the modules, difficult to understand. Both tests and systems can be modular. Testing can and should likewise be organised into modular components. Small, independent test cases can be designed to test independent modules.

- **Small Versus Large:** Programming in large means constructing programs that consists of many components written by many different programmers. Programming in the small is what we do for ourselves in the privacy of our own offices. Qualitative and Quantitative changes occur with size and so must testing methods and quality criteria.
- **Builder Versus Buyer:** Most software is written and used by the same organization. Unfortunately, this situation is dishonest because it clouds accountability. If there is no separation between builder and buyer, there can be no accountability.
- The different roles / users in a system include:
    1. **Builder:** Who designs the system and is accountable to the buyer.
    2. **Buyer:** Who pays for the system in the hope of profits from providing services.
    3. **User:** Ultimate beneficiary or victim of the system. The user's interests are also guarded by.
    4. **Tester:** Who is dedicated to the builder's destruction.
    5. **Operator:** Who has to live with the builders' mistakes, the buyers' murky (unclear) specifications, testers' oversights and the users' complaints.

Model for Testing - CO 1

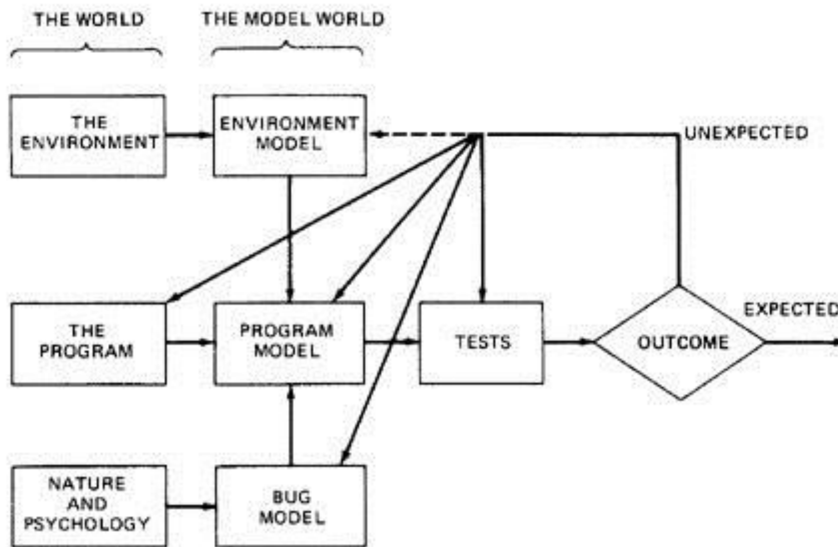


Figure : A Model for Testing

Above figure is a model of testing process. It includes three models: A model of the environment, a model of the program and a model of the expected bugs.

- **ENVIRONMENT:**
    - A Program's environment is the hardware and software required to make it run. For online systems, the environment may include communication lines, other systems, terminals and operators.
    - The environment also includes all programs that interact with and are used to create the program under test - such as OS, linkage editor, loader, compiler, utility routines.

- Because the hardware and firmware are stable, it is not smart to blame the environment for bugs.

- **PROGRAM:**
    - Most programs are too complicated to understand in detail.
    - The concept of the program is to be simplified 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:**
        1. **Benign Bug Hypothesis:** The belief that bugs are nice, tame and logical. (Benign: Not Dangerous)
        2. **Bug Locality Hypothesis:** The belief that a bug discovered with in a component effects only that component's behaviour.
        3. **Control Bug Dominance:** The belief that errors in the control structures (if, switch etc) of programs dominate the bugs.
        4. **Code / Data Separation:** The belief that bugs respect the separation of code and data.
        5. **Lingua Salvator Est:** The belief that the language syntax and semantics (e.g. Structured Coding, Strong typing, etc) eliminates most bugs.
        6. **Corrections Abide:** The mistaken belief that a corrected bug remains corrected.
        7. **Silver Bullets:** The mistaken belief that X (Language, Design method, representation, environment) grants immunity from bugs.
        8. **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.
        9. **Angelic Testers:** The belief that testers are better at test design than programmers are at code design.
- **TESTS:**
    - Tests are formal procedures, Inputs must be prepared, Outcomes should predicted, tests should be documented, commands need to be executed, and results are to be observed. *All these errors are subjected to error*
    - **We do three distinct kinds of testing on a typical software system. They are:**
        1. **Unit / Component Testing:** A **Unit** is the smallest testable piece of software that can be compiled, assembled, linked, loaded etc. A unit is usually the work of one programmer and consists of several hundred or fewer lines of code. **Unit Testing** is the testing we do to show that the unit does not satisfy its functional specification or that its implementation

structure does not match the intended design structure. A **Component** is an integrated aggregate of one or more units. **Component Testing** is the testing we do to show that the component does not satisfy its functional specification or that its implementation structure does not match the intended design structure.

2. **Integration Testing: Integration** is the process by which components are aggregated to create larger components. **Integration Testing** is testing done to show that even though the componenets were individually satisfactory (after passing component testing), checks the combination of components are incorrect or inconsistent.
3. **System Testing:** A **System** is a big component. **System Testing** is aimed at revealing bugs that cannot be attributed to components. It includes testing for performance, security, accountability, configuration sensitivity, startup and recovery.

- **Role of Models:** The art of testing consists of creating , selecting, exploring, and revising models. Our ability to go through this process depends on the number of different models we have at hand and their ability to express a program's behaviour.
- **PLAYING POOL AND CONSULTING ORACLES**
    - Testing is like playing a pool game. Either you hit the ball to any pocket (kiddie pool) or you specify the pocket in advance (real pool). So is the testing. There is kiddie testing and real testing. In kiddie testing, the observed outcome will be considered as the expected outcome. In Real testing, the outcome is predicted and documented before the test is run.
    - The tester who cannot make that kind of predictions does not understand the program's functional objectives.
    - **Oracles:** An oracle is any program, process, or body of data that specifies the expected outcome of a set of tests as applied to a tested object. Example of oracle : Input/Outcome Oracle - an oracle that specifies the expected outcome for a specified input.
    - **Sources of Oracles:** If every test designer had to analyze and predict the expected behaviour for every test case for every component, then test design would be very expensive. The hardest part of test design is predicting the expected outcome, but we often have oracles that reduce the work. They are:
        1. **Kiddie Testing:** run the test and see what comes out. If you have the outcome in front of you, and especially if you have the values of the internal variables, then it is much easier to validate that outcome by analysis and show it to be correct than it is to predict what the outcome should be and validate your prediction.
        2. **Regression Test Suites:** Today's software development and testing are dominated not by the design of new software but by rework and maintenance of existing software. In such instances, most of the tests you need will have been run on a previous versions. Most of those tests should have the same outcome for the new version. Outcome prediction is therefore needed only for changed parts of components.
        3. **Purchased Suits and Oracles:** Highly standardized software that differ only as to implementation often has commercially

available test suites and oracles. The most common examples are compilers for standard languages.

4. **Existing Program:** A working, trusted program is an excellent oracle. The typical use is when the program is being rehosted to a new language, operating system, environment, configuration with the intention that the behavior should not change as a result of the rehosting.

- **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.
- **THEORITICAL BARRIERS OF COMPLETE TESTING:**
    - "We can never be sure that the specifications are correct"
    - "No verification system can verify every correct program"
    - "We can never be certain that a verification system is correct"
- Not only all known approaches to absoulte demonstrations of correctness impractical, but they are impossible. Therefore, our objective must shift from a absolute proof to a 'suitably convincing' demonstration.

Consequences of Bugs - CO1

- **IMPORTANCE OF BUGS:** The importance of bugs depends on frequency, correction cost, installation cost, and consequences.
    1. **Frequency:** How often does that kind of bug occur? Pay more attention to the more frequent bug types.
    2. **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.
    3. **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.
    4. **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.

- **FLEXIBLE SEVERITY RATHER THAN ABSOLUTES:**

1. Quality can be measured as a combination of factors, of which number of bugs and their severity is only one component.
2. Many organizations have designed and used satisfactory, quantitative, quality metrics.
3. Because bugs and their symptoms play a significant role in such metrics, as testing progresses, you see the quality rise to a reasonable value which is deemed to be safe to ship the product.
4. The factors involved in bug severity are:
    1. **Correction Cost:** Not so important because catastrophic bugs may be corrected easier and small bugs may take major time to debug.
    2. **Context and Application Dependency:** Severity depends on the context and the application in which it is used.
    3. **Creating Culture Dependency:** Whats important depends on the creators of software and their cultural aspirations. Test tool vendors are more sensitive about bugs in their software then games software vendors.
    4. **User Culture Dependency:** Severity also depends on user culture. Naive users of PC software go crazy over bugs where as pros (experts) may just ignore.
    5. **The software development phase:** Severity depends on development phase. Any bugs gets more severe as it gets closer to field use and more severe the longer it has been around.

Taxonomy of Bugs - CO1

- 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:

1. Most feature bugs are rooted in human to human communication problems. One solution is to use high-level, formal specification languages or systems.
2. Such languages and systems provide short term support but in the long run, does not solve the problem.
3. ***Short term Support:*** Specification languages facilitate formalization of requirements and inconsistency and ambiguity analysis.
4. ***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.
5. 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:**

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

    - **External Interfaces:**
        - The external interfaces are the means used to communicate with the world.
        - These include devices, actuators, sensors, input terminals, printers, and communication lines.
        - The primary design criterion for an interface with outside world should be robustness.
        - All external interfaces, human or machine should employ a protocol. The protocol may be wrong or incorrectly implemented.
        - Other external interface bugs are: invalid timing or sequence assumptions related to external signals
        - Misunderstanding external input or output formats.
        - Insufficient tolerance to bad input data.
    - **Internal Interfaces:**
        - Internal interfaces are in principle not different from external interfaces but they are more controlled.
        - A best example for internal interfaces are communicating routines.
        - The external environment is fixed and the system must adapt to it but the internal environment, which consists of interfaces with other components, can be negotiated.
        - Internal interfaces have the same problem as external interfaces.
    - **Hardware Architecture:**
        - Bugs related to hardware architecture originate mostly from misunderstanding how the hardware works.
        - Examples of hardware architecture bugs: address generation error, i/o device operation / instruction error, waiting too long for a response, incorrect interrupt handling etc.
        - The remedy for hardware architecture and interface problems is two fold: (1) Good Programming and Testing (2) Centralization of hardware interface software in programs written by hardware interface specialists.
    - **Operating System Bugs:**

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

- **Software Architecture:**
  - Software architecture bugs are the kind that called - interactive.
  - Routines can pass unit and integration testing without revealing such bugs.
  - Many of them depend on load, and their symptoms emerge only when the system is stressed.
  - Sample for such bugs: Assumption that there will be no interrupts, Failure to block or un block interrupts, Assumption that memory and registers were initialized or not initialized etc
  - Careful integration of modules and subjecting the final system to a stress test are effective methods for these bugs.
- **Control and Sequence Bugs (Systems Level):**
  - These bugs include: Ignored timing, Assuming that events occur in a specified sequence, Working on data before all the data have arrived from disc, Waiting for an impossible combination of prerequisites, Missing, wrong, redundant or superfluous process steps.
  - The remedy for these bugs is highly structured sequence control.
  - Specialize, internal, sequence control mechanisms are helpful.
- **Resource Management Problems:**
  - Memory is subdivided into dynamically allocated resources such as buffer blocks, queue blocks, task control blocks, and overlay buffers.
  - External mass storage units such as discs, are subdivided into memory resource pools.
  - Some resource management and usage bugs: Required resource not obtained,

Wrong resource used, Resource is already in use, Resource dead lock etc

    - **Resource Management Remedies:** A design remedy that prevents bugs is always preferable to a test method that discovers them.
    - The design remedy in resource management is to keep the resource structure simple: the fewest different kinds of resources, the fewest pools, and no private resource management.
- **Integration Bugs:**
    - Integration bugs are bugs having to do with the integration of, and with the interfaces between, working and tested components.
    - These bugs results from inconsistencies or incompatibilities between components.
    - The communication methods include data structures, call sequences, registers, semaphores, communication links and protocols results in integration bugs.
    - The integration bugs do not constitute a big bug category(9%) they are expensive category because they are usually caught late in the game and because they force changes in several components and/or data structures.
- **System Bugs:**
    - System bugs covering all kinds of bugs that cannot be ascribed to a component or to their simple interactions, but result from the totality of interactions between many components such as programs, data, hardware, and the operating systems.
    - There can be no meaningful system testing until there has been thorough component and integration testing.
    - System bugs are infrequent(1.7%) but very important because they are often found only after the system has been fielded.

2. **TEST AND TEST DESIGN BUGS:**

- Testing: testers have no immunity to bugs. Tests require complicated scenarios and databases.

- They require code or the equivalent to execute and consequently they can have bugs.
- Test criteria: if the specification is correct, it is correctly interpreted and implemented, and a proper test has been designed; but the criterion by which the software's behavior is judged may be incorrect or impossible. So, a proper test criteria has to be designed. The more complicated the criteria, the likelier they are to have bugs.
- **Remedies:** The remedies of test bugs are:

  - **Test Debugging:** The first remedy for test bugs is testing and debugging the tests. Test debugging, when compared to program debugging, is easier because tests, when properly designed are simpler than programs and donot have to make concessions to efficiency.
  - **Test Quality Assurance:** Programmers have the right to ask how quality in independent testing is monitored.
  - **Test Execution Automation:** The history of software bug removal and prevention is indistinguishable from the history of programming automation aids. Assemblers, loaders, compilers are developed to reduce the incidence of programming and operation errors. Test execution bugs are virtually eliminated by various test execution automation tools.
  - **Test Design Automation:** Just as much of software development has been automated, much test design can be and has been automated. For a given productivity rate, automation reduces the bug count - be it for software or be it for tests.

Overview of Unit and Integration Testing - CO1

Unit Tests are conducted by developers and test the unit of code(aka module, component) he or she developed. It is a testing method by which individual units of source code are tested to determine if they are ready to use. It helps to reduce the cost of bug fixes since the bugs are identified during the early phases of development life cycle.

Integration testing is executed by testers and tests integration between software modules. It is a software testing technique where individual units of a program are combined and tested as a group. Test stubs and test drivers are used to assist in Integration Testing. Integration test is performed in two way, they are bottom-up method and the top-down method.

Below is a detailed comparison between the two-

Unit test	Integration test
• The idea behind Unit Testing is to test each part of the program and show that the individual parts are correct.	• The idea behind Integration Testing is to combine modules in the application and test as a group to see that they are working fine
• It is kind of White Box Testing	• It is kind of Black Box Testing
• It can be performed at any time	• It usually carried out after Unit Testing and before System Testing
• Unit Testing tests only the functionality of the units themselves and may not catch integration errors, or other system-wide issues	• Integrating testing may detect errors when modules are integrated to build the overall system
• It starts from the module	• It starts from the interface

specification	specification
• It pays attention to the behavior of single modules	• It pays attention to integration among modules
• Unit test does not verify whether your code works with external dependencies correctly.	• Integration tests verifies that your code works with external dependencies correctly.
• It is usually executed by developer	• It is usually executed by test team
• Finding errors are easy	• Finding errors are difficult
• Maintenance of unit test is cheap	• Maintenance of integration test is expensive

CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 1
Unit - 2
Year and Semester: IV yr & I Sem
Subject**: Software Testing Methodologies**
Branch: **CSE**
Faculty: **Revathi Lavanya Baggam,** Assistant Professor (CSE)

Basic concepts of path testing - CO2

- **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 indespensable.
- **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 are executed.
- Formally, a process block is a piece of straight line code of one statement or hundreds of statements.
- A process has one entry and one exit. It can consists of a single statement or instruction, a sequence of statements or instructions, a single entry/exit subroutine, a macro or function call, or a sequence of these.

2. **Decisions:**
    - A decision is a program point at which the control flow can diverge.
    - Machine language conditional branch and conditional skip instructions are examples of decisions.
    - Most of the decisions are two-way but some are three way branches in control flow.
3. **Case Statements:**
    - A case statement is a multi-way branch or decisions.
    - Examples of case statement are a jump table in assembly language, and the PASCAL case statement.
    - From the point of view of test design, there are no differences between Decisions and Case Statements
4. **Junctions:**
    - A junction is a point in the program where the control flow can merge.
    - Examples of junctions are: the target of a jump or skip instruction in ALP, a label that is a target of GOTO.

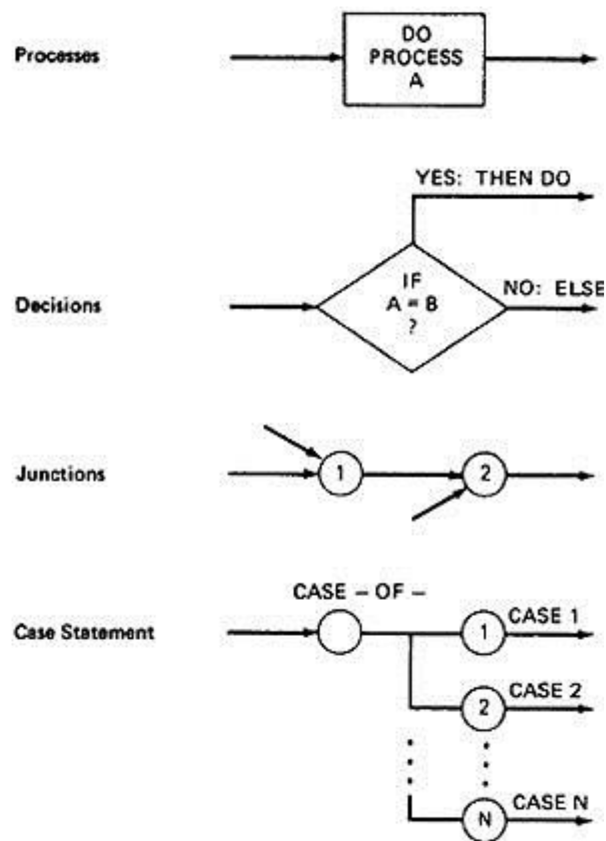


Figure 2.1: Flowgraph Elements

1. **CONTROL FLOW GRAPHS Vs FLOWCHARTS:**
   - A program's flow chart resembles a control flow graph.
   - In flow graphs, we don't show the details of what is in a process block.
   - In flow charts every part of the process block is drawn.
   - The flowchart focuses on process steps, where as the flow graph focuses on control flow of the program.
   - The act of drawing a control flow graph is a useful tool that can help us clarify the control flow and data flow issues.
2. **NOTATIONAL EVOULTION:**
   - The control flow graph is simplified representation of the program's structure.
   - The notation changes made in creation of control flow graphs:

     1. The process boxes weren't really needed. There is an implied process on every line joining junctions and decisions.
     2. We don't need to know the specifics of the decisions, just the fact that there is a branch.
     3. The specific target label names aren't important-just the fact that they exist. So we can replace them by simple numbers.

4. To understand this, we will go through an example (Figure 2.2) written in a FORTRAN like programming language called **Programming Design Language (PDL)**. The program's corresponding flowchart (Figure 2.3) and flowgraph (Figure 2.4) were also provided below for better understanding.
5. The first step in translating the program to a flowchart is shown in Figure 2.3, where we have the typical one-for-one classical flowchart. Note that complexity has increased, clarity has decreased, and that we had to add auxiliary labels (LOOP, XX, and YY), which have no actual program counterpart. In Figure 2.4 we merged the process steps and replaced them with the single process box. We now have a control flowgraph. But this representation is still too busy. We simplify the notation further to achieve Figure 2.5, where for the first time we can really see what the control flow looks like.

CODE* (PDL)

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

* A contrived horror

Figure 2.2: Program Example (PDL)

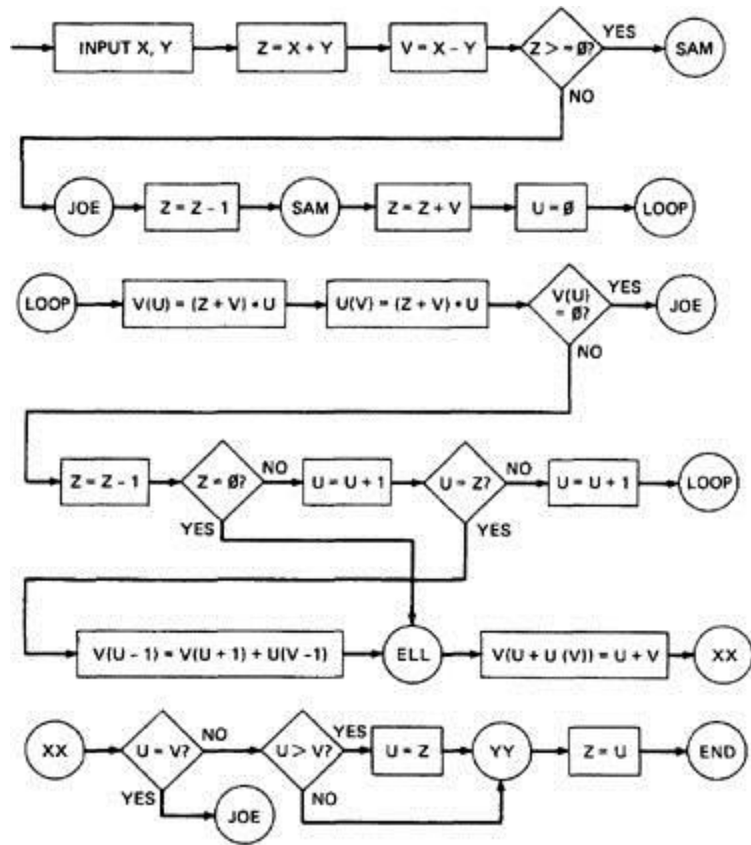


Figure 2.3: One-to-one flowchart for example program in Figure 2.2

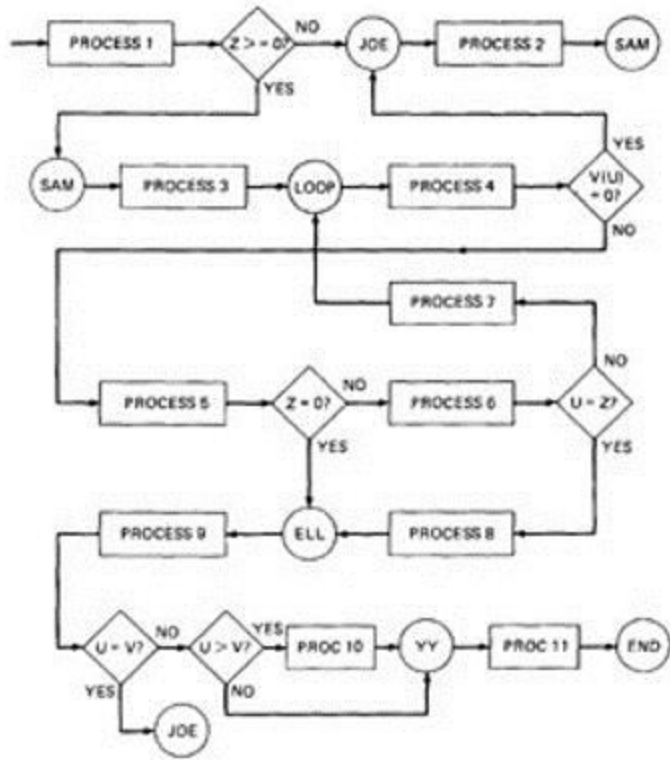


Figure 2.4: Control Flowgraph for example in Figure 2.2

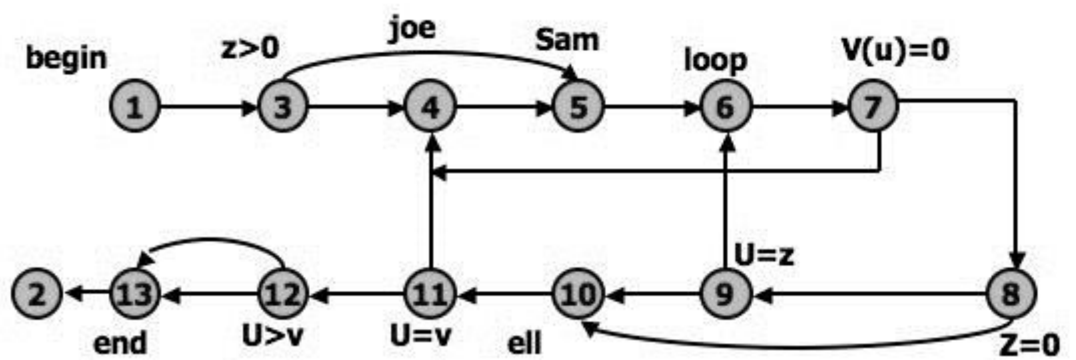


Figure 2.5: Simplified Flowgraph Notation

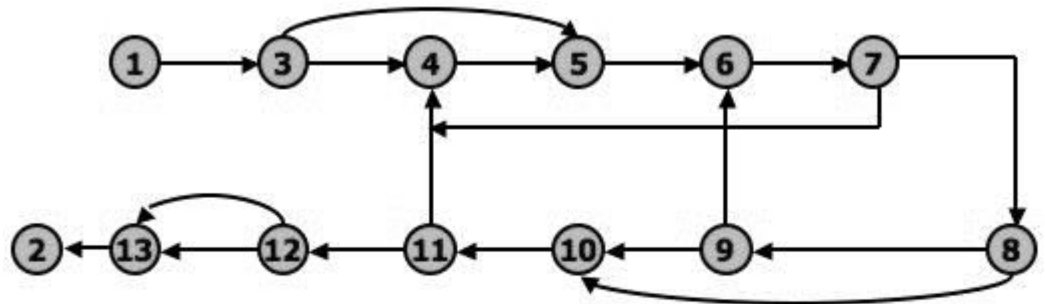


Figure 2.6: Even Simplified Flowgraph Notation

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

3.

- **LINKED LIST REPRESENTATION:**

1. Although graphical representations of flowgraphs are revealing, the details of the control flow inside a program they are often inconvenient.
2. 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.
3. **Linked List representation of Flow Graph:**

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

Figure 2.7: Linked List Control Flowgraph Notation

- **FLOWGRAPH - PROGRAM CORRESPONDENCE:**

1. A flow graph is a pictorial representation of a program and not the program itself, just as a topographic map.
2. You cant always associate the parts of a program in a unique way with flowgraph parts because many program structures, such as if-then-else constructs, consists of a combination of decisions, junctions, and processes.
3. The translation from a flowgraph element to a statement and vice versa is not always unique. (See Figure 2.8)

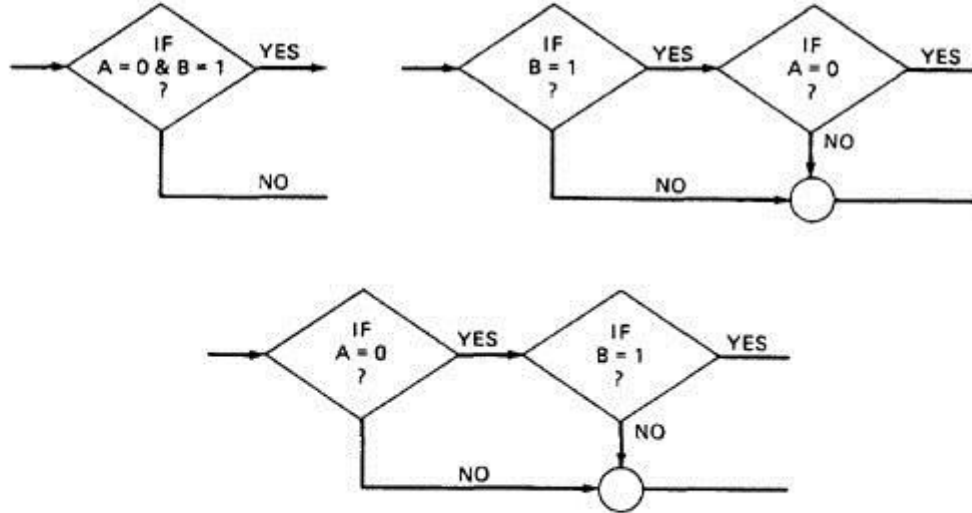


Figure 2.8: Alternative Flowgraphs for same logic (Statement "IF (A=0) AND (B=1) THEN . . .").

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

- **FLOWGRAPH AND FLOWCHART GENERATION:**

1. Flowcharts can be

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

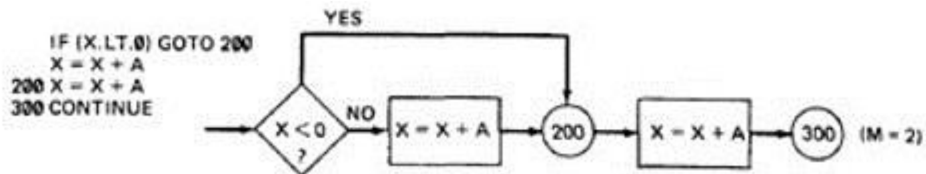
2. There are relatively few control flow graph generators.

   2. **PATH TESTING - PATHS, NODES AND LINKS:**

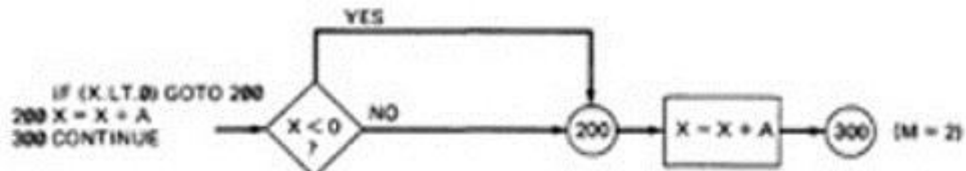
1. **Path:**a path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same junction, decision, or exit.
2. A path may go through several junctions, processes, or decisions, one or more times.
3. Paths consists of segments.
4. The segment is a link - a single process that lies between two nodes.
5. A path segment is succession of consecutive links that belongs to some path.
6. 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.
7. The name of a path is the name of the nodes along the path.

   3. **FUNDAMENTAL PATH SELECTION CRITERIA:**

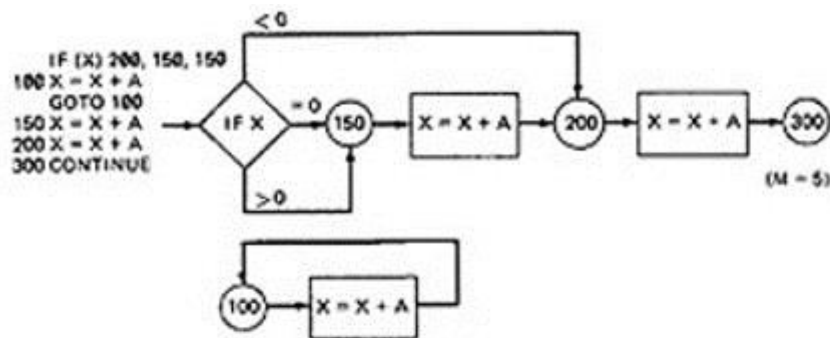
1. There are many paths between the entry and exit of a typical routine.
2. Every decision doubles the number of potential paths. And every loop multiplies the number of potential paths by the number of different iteration values possible for the loop.
3. Defining complete testing:
    1. Exercise every path from entry to exit
    2. Exercise every statement or instruction at least once
    3. Exercise every branch and case statement, in each direction at least once
4. 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.
5. ***EXAMPLE:*Here is the correct version.**



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



A negative value produces the correct answer. Every statement can be executed, but if the test cases do not force each branch to be taken, the bug can remain hidden. The next example uses a test based on executing each branch but does not force the execution of all statements:



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

6. 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.
7. 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.

4. **PATH TESTING CRITERIA:**

1. Any testing strategy based on paths must at least both exercise every instruction and take branches in all directions.
2. 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.
3. So we have explored three different testing criteria or strategies out of a potentially infinite family of strategies.
    1. **Path Testing (P_{inf})**:
        - Execute all possible control flow paths through the program: typically, this is restricted to all possible entry/exit paths through the program.
        - If we achieve this prescription, we are said to have achieved 100% path coverage. This is the strongest criterion in the path testing strategy family: it is generally impossible to achieve.
    2. **Statement Testing (P_1)**:
        - Execute all statements in the program at least once under some test. If we do enough tests to achieve

this, we are said to have achieved 100% statement coverage.
- An alternate equivalent characterization is to say that we have achieved 100% node coverage. We denote this by C1.
- This is the weakest criterion in the family: testing less than this for new software is unconscionable (unprincipled or can not be accepted) and should be criminalized.

3. **Branch Testing (P_2)**:
   - Execute enough tests to assure that every branch alternative has been exercised at least once under some test.
   - If we do enough tests to achieve this prescription, then we have achieved 100% branch coverage.
   - An alternative characterization is to say that we have achieved 100% link coverage.
   - For structured software, branch testing and therefore branch coverage strictly includes statement coverage.
   - We denote branch coverage by C2.
   - **Commonsense and Strategies:**

1. Branch and statement coverage are accepted today as the minimum mandatory testing requirement.
2. 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.
3. **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.
4. **Path Selection Example:**

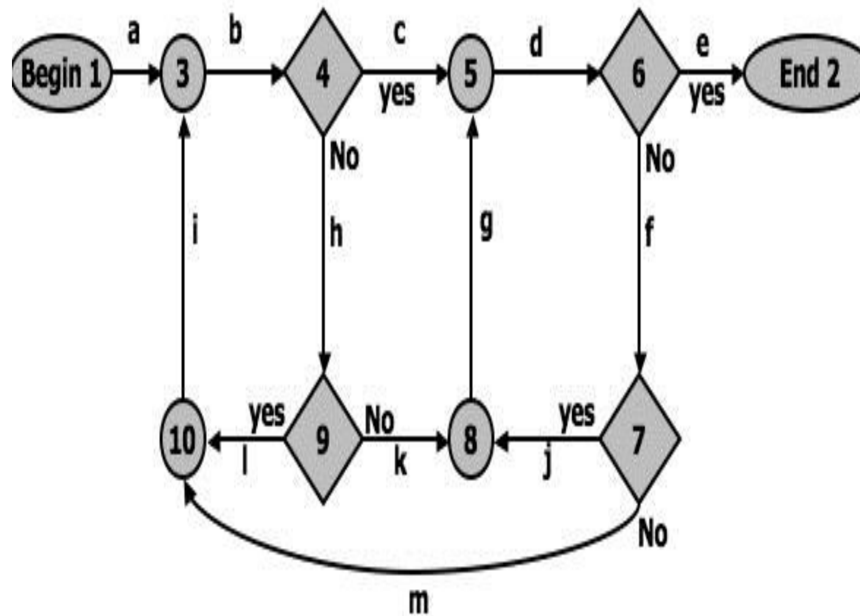


Figure 2.9: An example flowgraph to explain path selection

5. **Practical Suggestions in Path Testing:**

   - Draw the control flow graph on a single sheet of paper.
   - Make several copies - as many as you will need for coverage (C1+C2) and several more.
   - Use a yellow highlighting marker to trace paths. Copy the paths onto a master sheets.
   - Continue tracing paths until all lines on the master sheet are covered, indicating that you appear to have achieved C1+C2.
   - As you trace the paths, create a table that shows the paths, the coverage status of each process, and each decision.
   - The above paths lead to the following table considering Figure 2.9:

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

- After you have traced a a covering path set on the master sheet and filled in the table for 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)
- **Revised Path Selection Rules:**

    1. Pick the simplest, functionally sensible entry/exit path.
    2. Pick additional paths as small variation from previous paths. Pick paths that do not have loops rather than paths that do. Favor short paths that make sense over paths that don't.
    3. Pick additional paths that have no obvious functional meaning only if it's necessary to provide coverage.
    4. Be comfortable with your chosen paths. Play your hunches (guesses) and give

your intuition free reign as long as you achieve C1+C2.
5. Don't follow rules slavishly (blindly) - except for coverage.

2. **LOOPS:**
   - **Cases for a single loop:**A Single loop can be covered with two cases: Looping and Not looping. But, experience shows that many loop-related bugs are not discovered by C1+C2. Bugs hide themselves in corners and congregate at boundaries - in the cases of loops, at or around the minimum or maximum number of times the loop can be iterated. The minimum number of iterations is often zero, but it need not be.
     CASE 1: Single loop, Zero minimum, N maximum, No excluded values

- Try bypassing the loop (zero iterations). If you can't, you either have a bug, or zero is not the minimum and you have the wrong case.
- Could the loop-control variable be negative? Could it appear to specify a negative number of iterations? What happens to such a value?
- One pass through the loop.
- Two passes through the loop.
- A typical number of iterations, unless covered by a previous test.
- One less than the maximum number of iterations.
- The maximum number of iterations.
- Attempt one more than the maximum number of iterations. What prevents the loop-control variable from having this value? What will happen with this value if it is forced?

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

- Try one less than the expected minimum. What happens if the loop control variable's value is less than the minimum? What prevents the value from being less than the minimum?
- The minimum number of iterations.
- One more than the minimum number of iterations.
- Once, unless covered by a previous test.
- Twice, unless covered by a previous test.
- A typical value.
- One less than the maximum value.
- The maximum number of iterations.
- Attempt one more than the maximum number of iterations.

CASE 3: Single loops with excluded values

- Treat single loops with excluded values as two sets of tests consisting of loops without excluded values, such as case 1 and 2 above.
- Example, the total range of the loop control variable was 1 to 20, but that values 7,8,9,10 were excluded. The two sets of tests are 1-6 and 11-20.
- The test cases to attempt would be 0,1,2,4,6,7 for the first range and 10,11,15,19,20,21 for the second range.
  - **Kinds of Loops:**There are only three kinds of loops with respect to path testing:
- **Nested Loops:**
  1. The number of tests to be performed on nested loops will be the exponent of the tests performed on single loops.
  2. 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:**
  1. 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.

2. If the loops cannot be on the same path, then they are not concatenated and can be treated as individual loops.

- **Horrible Loops:**
  1. 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.
  2. Makes iteration value selection for test cases an awesome and ugly task, which is another reason such structures should be avoided.

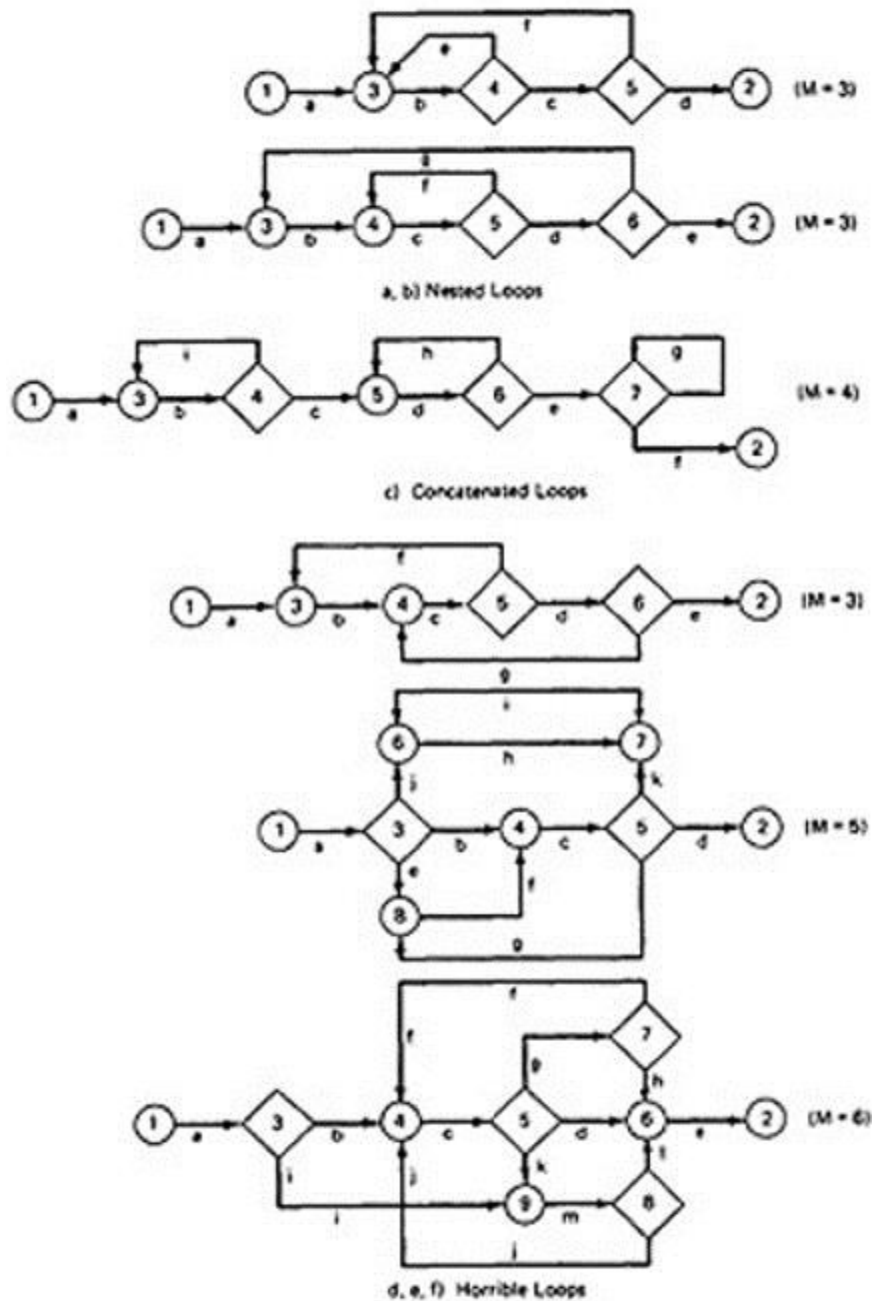


Figure 2.10: Example of Loop types

- **Loop Testing TIme:**

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

- This situation is obviously worse for nested and dependent concatenated loops.
- Consider nested loops in which testing the combination of extreme values lead to long test times. Several options to deal with:
    1. Prove that the combined extreme cases are hypothetically possible, they are not possible in the real world
    2. 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 -

CO2

- **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 assignement 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.**
    - Some times the interpretation may depend on the path; for example,
    - `INPUT X`
    - `ON X GOTO A, B, C, ...`
    - `A: Z := 7 @ GOTO HEM`
    - `B: Z := -7 @ GOTO HEM`
    - `C: Z := 0 @ GOTO HEM`
    - `.........`
    - `HEM: DO SOMETHING`
    - `.........`
    - `HEN: IF Y + Z > 0 GOTO ELL ELSE GOTO EMM`

The predicate interpretation at HEN depends on the path we took through the first multiway branch. It yields for the three cases respectively, if Y+7>0, Y-7>0, Y>0.

    - The path predicates are the specific form of the predicates of the decisions along the selected path after interpretation.
- **INDEPENDENCE OF VARIABLES AND PREDICATES:**
    - The path predicates take on truth values based on the values of input variables, either directly or indirectly.
    - If a variable's value does not change as a result of processing, that variable is independent of the processing.

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

- **CORRELATION OF VARIABLES AND PREDICATES:**
  - Two variables are correlated if every combination of their values cannot be independently specified.
  - Variables whose values can be specified independently without restriction are called uncorrelated.
  - A pair of predicates whose outcomes depend on one or more variables in common are said to be correlated predicates.
    For example, the predicate X==Y is followed by another predicate X+Y == 8. If we select X and Y values to satisfy the first predicate, we might have forced the 2nd predicate's truth value to change.
  - Every path through a routine is achievable only if all the predicates in that routine are uncorrelated.
- **PATH PREDICATE EXPRESSIONS:**
  - A path predicate expression is a set of boolean expressions, all of which must be satisfied to achieve the selected path.
  - Example:
  - `X1+3X2+17>=0`
  - `X3=17`
  - `X4-X1>=14X2`
  - Any set of input values that satisfy all of the conditions of the path predicate expression will force the routine to the path.
  - 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	C: X3 = 17
D: X4 - X1 >= 14X2	D: X4 - X1 >= 14X2

-
- Boolean algebra notation to denote the boolean expression:

ABCD+EBCD=(A+E)BCD

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

- **Assignment Blindness:**
  1. 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.
  2. For Example:

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

  3. If the test case sets Y=1 the desired path is taken in either case, but there is still a bug.
- **Equality Blindness:**
  1. Equality blindness occurs when the path selected by a prior predicate results in a value that works both for the correct and buggy predicate.
  2. For Example:

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

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

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

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

3. 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 - CO2

2. **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+BCEFGHIJK L

- 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**.

3. **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 - CO2

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

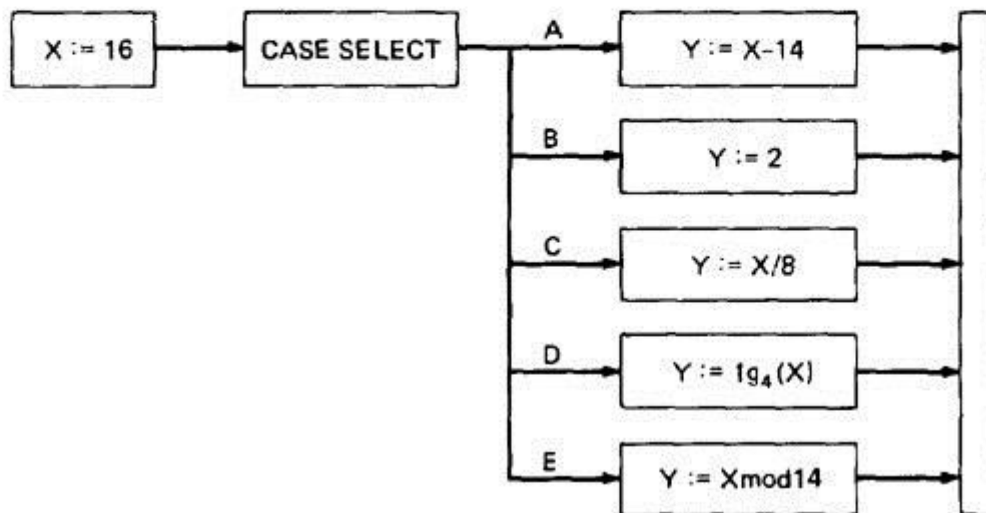
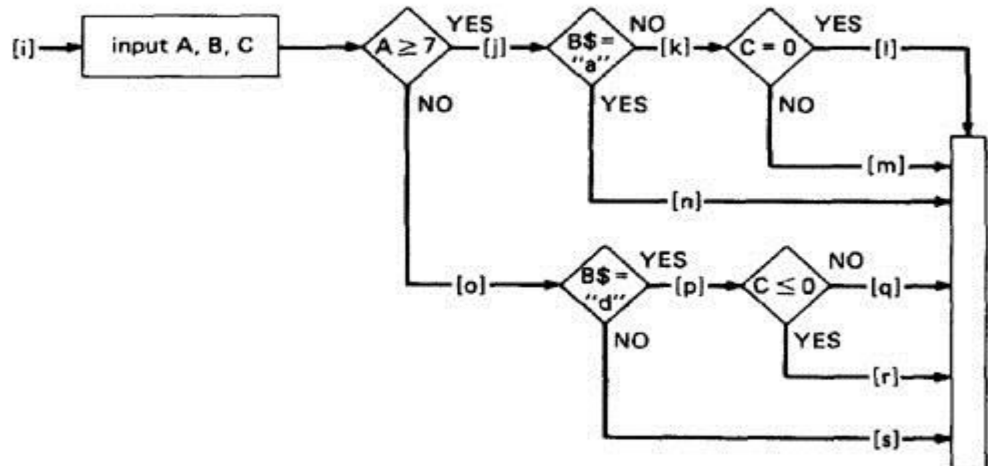


Figure 2.11: Coincidental Correctness

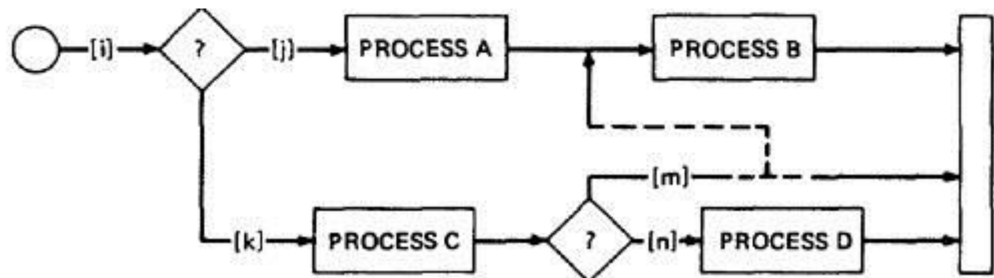
The above figure is an example of a routine that, for the (unfortunately) chosen input value (X = 16), yields the same outcome (Y = 2) no matter which case we select. Therefore, the tests chosen this way will not tell us whether we have achieved coverage. For example, the five cases could be totally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

- The types of instrumentation methods include:
  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.

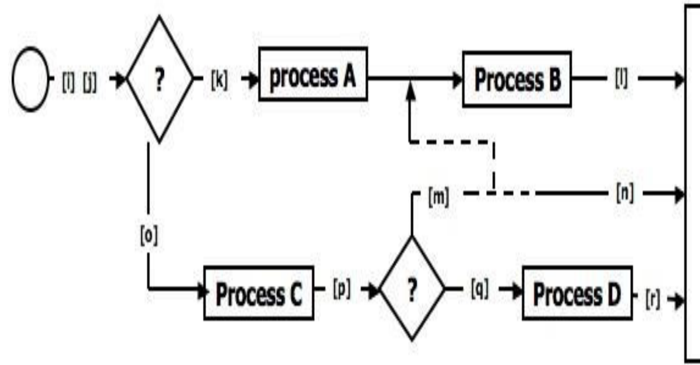


3. ***Figure 2.12: Single Link Marker Instrumentation***

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



4. ***Figure 2.13: Why Single Link Markers aren't enough.***
5. 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.
6. **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.



7. ***Figure 2.14: Double Link Marker Instrumentation.***
8. **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.

APPLICATION OF PATH TESTING - CO2

Integration, Coverage and Paths in called Components

Path testing methods are mainly used in unit testing, especially for new software

The new component is first tested as an independent unit with all called components and co-requisite components replaced by stubs. A simulator of low-level components that is more reliable than the actual component

Path testing clarifies the integration issues

C_1 coverage at the system level ranges from 50% to 85%

We gave no statistics for C_2 coverage in system testing because it is impossible to monitor C2 coverage without disrupting the system's operation

Note:

Co-requisite: A formal course of study required to be taken simultaneously with another

New Code:

New code should always be subjected to enough path testing to achieve C_2

Stubs are used where it is clear that the bug potential for the stub is significantly lower than that of the called components

Old, trusted components will not be replaced by stubs

Some consideration is given to paths within called components

Typically, we will try to use the shortest entry/exit path that will do the task

Maintenance:

There is a great difference between maintenance testing and new code testing

Maintenance testing is a completely different situation

It involves modifications which are accommodated in the system, as required
Path testing is used firstly on the modified component

Rehosting:
Path testing with C_1+C_2 coverage is a powerful tool for rehosting old software
We get a very powerful, effective, rehosting process when C_1+C_2 coverage is used in conjunction with automatic or semiautomatic structural test generators
Software is rehosted because it is no longer cost effective to support the environment in which it runs
The objective of rehosting is to change the operating environment and not the rehosted software
Rehosting from one COBOL environment to another is easy by comparison
Rehosted software can be modified to improve efficiency and/or to implement new functionality, which had been difficult in the old environments
The test suites(collection) and all outcomes of the old environment become the specification for the rehosted software

PATH, PATH PRODUCTS AND REGULAR EXPRESSIONS - CO2

- **MOTIVATION:**
  - Flow graphs are being an abstract representation of programs.
  - Any question about a program can be cast into an equivalent question about an appropriate flowgraph.
  - Most software development, testing and debugging tools use flow graphs analysis techniques.
- **PATH PRODUCTS:**
  - Normally flow graphs used to denote only control flow connectivity.
  - The simplest weight we can give to a link is a name.
  - Using link names as weights, we then convert the graphical flow graph into an equivalent algebraic like expressions which denotes the set of all possible paths from entry to exit for the flow graph.
  - Every link of a graph can be given a name.
  - The link name will be denoted by lower case italic letters.
  - In tracing a path or path segment through a flow graph, you traverse a succession of link names.
  - The name of the path or path segment that corresponds to those links is expressed naturally by concatenating those link names.
  - For example, if you traverse links a,b,c and d along some path, the name for that path segment is abcd. This path name is also called a **path product.** Figure 5.1 shows some examples:

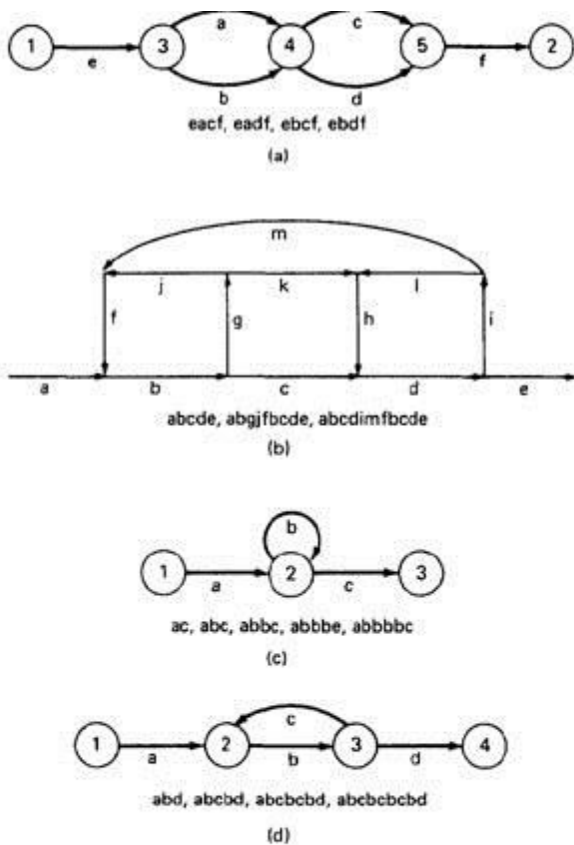


Figure 5.1: Examples of paths.

- **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

- If a link or segment name is repeated, that fact is denoted by an exponent. The exponent's value denotes the number of repetitions:
- a^1 = a; a^2 = aa; a^3 = aaa; a^n = aaaa . . . n times.

  Similarly, if

  X = abcde

  then

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

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

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

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

- The zeroth power of a link name, path product, or path expression is also needed for completeness. It is denoted by the numeral "1" and denotes the "path" whose length is zero - that is, the path that doesn't have any links.
- $a^0 = 1$
- $X^0 = 1$

- **PATH SUMS:**
  - The "+" sign was used to denote the fact that path names were part of the same set of paths.
  - The "PATH SUM" denotes paths in parallel between nodes.
  - Links a and b in Figure 5.1a are parallel paths and are denoted by a + b. Similarly, links c and d are parallel paths between the next two nodes and are denoted by c + d.
  - The set of all paths between nodes 1 and 2 can be thought of as a set of parallel paths and denoted by eacf+eadf+ebcf+ebdf.
  - If X and Y are sets of paths that lie between the same pair of nodes, then X+Y denotes the UNION of those set of paths. For example, in Figure 5.2:

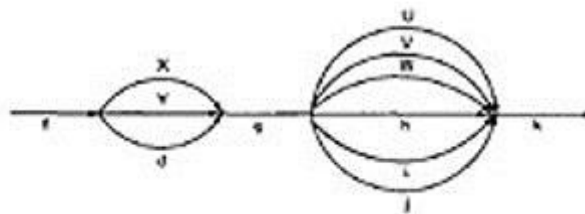


Figure 5.2: Examples of path sums.

The first set of parallel paths is denoted by X + Y + d and the second set by U + V + W + h + i + j. The set of all paths in this flowgraph is f(X + Y + d)g(U + V + W + h + i + j)k

  - The path is a set union operation, it is clearly Commutative and Associative.
  - RULE 2: X+Y=Y+X
  - RULE 3: (X+Y)+Z=X+(Y+Z)=X+Y+Z

- **DISTRIBUTIVE LAWS:**

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

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

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

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

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

  - If a set consists of paths names and a member of that set is added to it, the "new" name, which is already in that set of names, contributes nothing and can be ignored.
  - For example,
  - if X=a+aa+abc+abcd+def then
    X+a = X+aa = X+abc = X+abcd = X+def = X

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

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

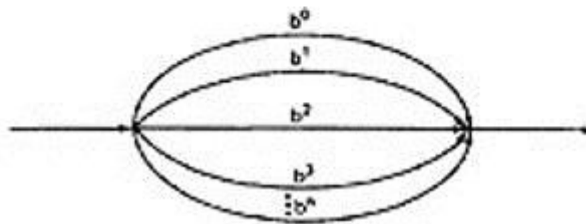


Figure 5.3: Examples of path loops.

  - This potentially infinite sum is denoted by b* for an individual link and by X* when X is a path expression.

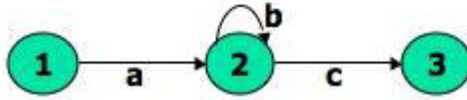


Figure 5.4: Another example of path loops.

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

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

- Evidently,

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

- It is more convenient to denote the fact that a loop cannot be taken more than a certain, say n, number of times.
- A bar is used under the exponent to denote the fact as follows:

  $X^{\underline{n}} = X^0+X^1+X^2+X^3+X^4+X^5+..................+X^n$

- **RULES 6 - 16:**
  - The following rules can be derived from the previous rules:
  - RULE 6: $X^{\underline{n}} + X^{\underline{m}} = X^{\underline{n}}$ if n>m

    RULE 6: $X^{\underline{n}} + X^{\underline{m}} = X^{\underline{m}}$ if m>n

    RULE 7: $X^n X^m = X^{\underline{n+m}}$

    RULE 8: $X^n X^* = X^* X^n = X^*$

    RULE 9: $X^n X^+ = X^+ X^n = X^+$

    RULE 10: $X^* X^+ = X^+ X^* = X^+$

    RULE 11: $1 + 1 = 1$

    RULE 12: $1X = X1 = X$

    Following or preceding a set of paths by a path of zero length does not change the set.

    RULE 13: $1^n = 1^{\underline{n}} = 1^* = 1^+ = 1$

    No matter how often you traverse a path of zero length,It is a path of zero length.

    RULE 14: $1^+ +1 = 1^*=1$

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

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

    RULE 16: $0X=X0=0$

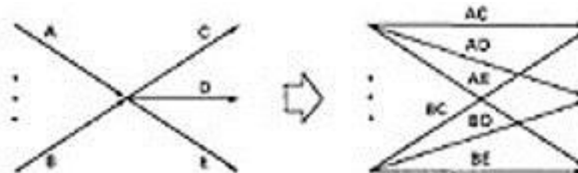
    If you block the paths of a graph for or aft by a graph that has no paths , there wont be any paths.

REDUCTION PROCEDURE - CO2

- **REDUCTION PROCEDURE ALGORITHM:**
  - This section presents a reduction procedure for converting a flowgraph whose links are labeled with names into a path expression that denotes the set of all entry/exit paths in that flowgraph. The procedure is a node-by-node removal algorithm.
  - The steps in Reduction Algorithm are as follows:
    1. Combine all serial links by multiplying their path expressions.
    2. Combine all parallel links by adding their path expressions.
    3. Remove all self-loops (from any node to itself) by replacing them with a link of the form X*, where X is the path expression of the link in that loop.

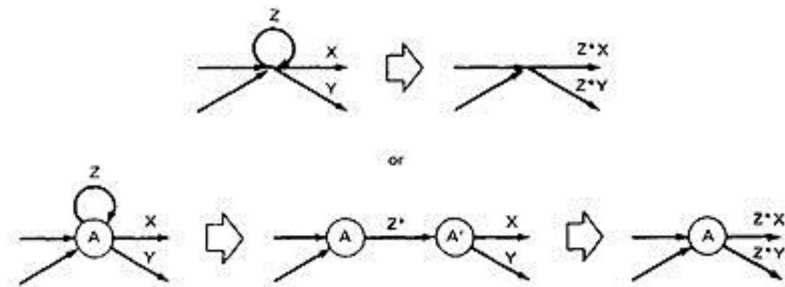
    STEPS 4 - 8 ARE IN THE ALGORIHTM'S LOOP:

    4. Select any node for removal other than the initial or final node. Replace it with a set of equivalent links whose path expressions correspond to all the ways you can form a product of the set of inlinks with the set of outlinks of that node.
    5. Combine any remaining serial links by multiplying their path expressions.
    6. Combine all parallel links by adding their path expressions.
    7. Remove all self-loops as in step 3.
    8. Does the graph consist of a single link between the entry node and the exit node? If yes, then the path expression for that link is a path expression for the original flowgraph; otherwise, return to step 4.
  - A flowgraph can have many equivalent path expressions between a given pair of nodes; that is, there are many different ways to generate the set of all paths between two nodes without affecting the content of that set.
  - The appearance of the path expression depends, in general, on the order in which nodes are removed.
- **CROSS-TERM STEP (STEP 4):**
  - The cross - term step is the fundamental step of the reduction algorithm.
  - It removes a node, thereby reducing the number of nodes by one.
  - Successive applications of this step eventually get you down to one entry and one exit node. The following diagram shows the situation at an arbitrary node that has been selected for removal:



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

- **LOOP REMOVAL OPERATIONS:**
  - There are two ways of looking at the loop-removal operation:



  - In the first way, we remove the self-loop and then multiply all outgoing links by Z*.
  - In the second way, we split the node into two equivalent nodes, call them A and A' and put in a link between them whose path expression is Z*. Then we remove node A' using steps 4 and 5 to yield outgoing links whose path expressions are Z*X and Z*Y.
- **A REDUCTION PROCEDURE - EXAMPLE:**
  - Let us see by applying this algorithm to the following graph where we remove several nodes in order; that is

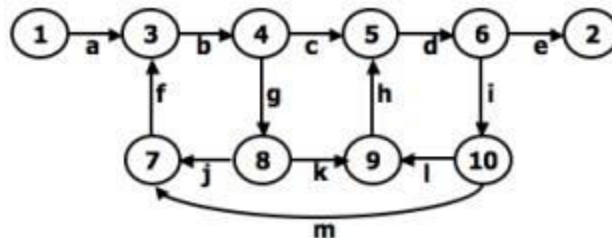
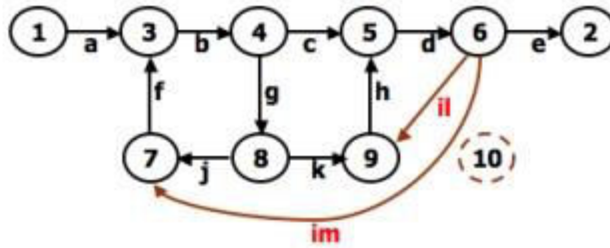
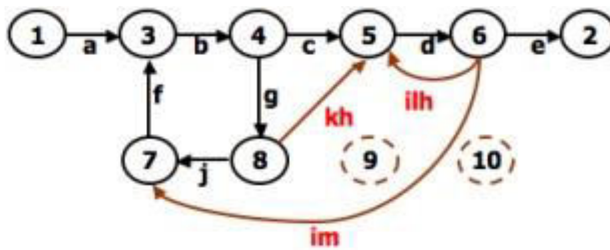


Figure 5.5: Example Flowgraph for demonstrating reduction procedure.

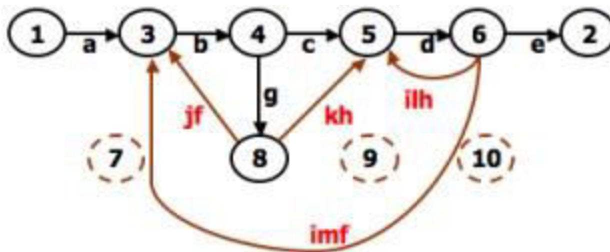
- Remove node 10 by applying step 4 and combine by step 5 to yield



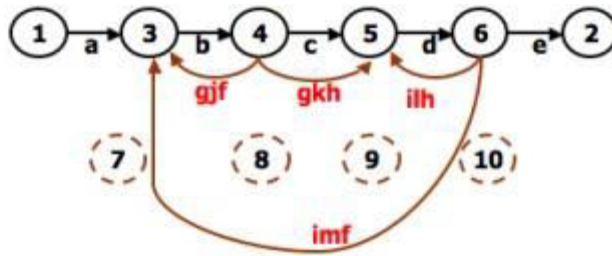
- Remove node 9 by applying step4 and 5 to yield



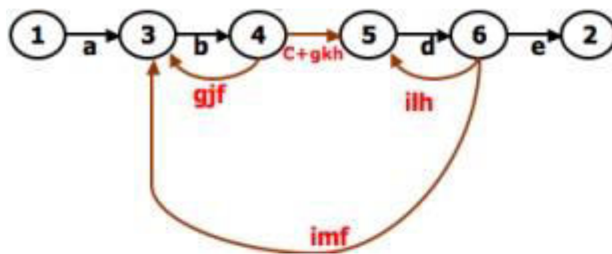
- Remove node 7 by steps 4 and 5, as follows:



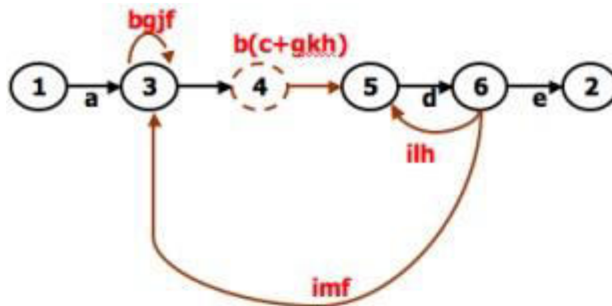
- Remove node 8 by steps 4 and 5, to obtain:



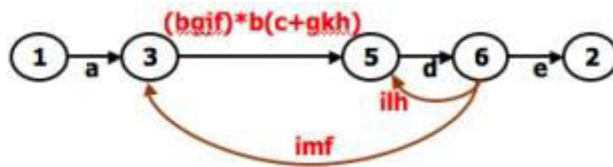
- **PARALLEL TERM (STEP 6):** Removal of node 8 above led to a pair of parallel links between nodes 4 and 5. combine them to create a path expression for an equivalent link whose path expression is c+gkh; that is



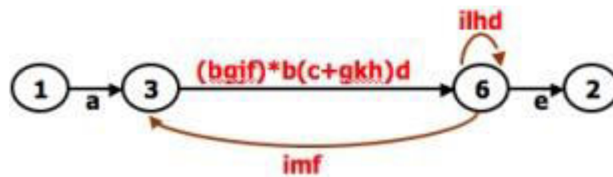
- **LOOP TERM (STEP 7):** Removing node 4 leads to a loop term. The graph has now been replaced with the following equivalent simpler graph:



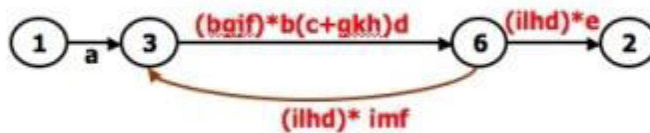
- Continue the process by applying the loop-removal step as follows:



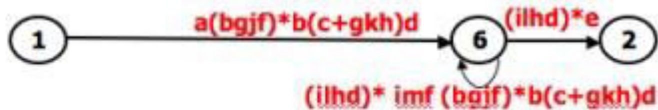
- Removing node 5 produces:



- Remove the loop at node 6 to yield:



- Remove node 3 to yield:



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

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

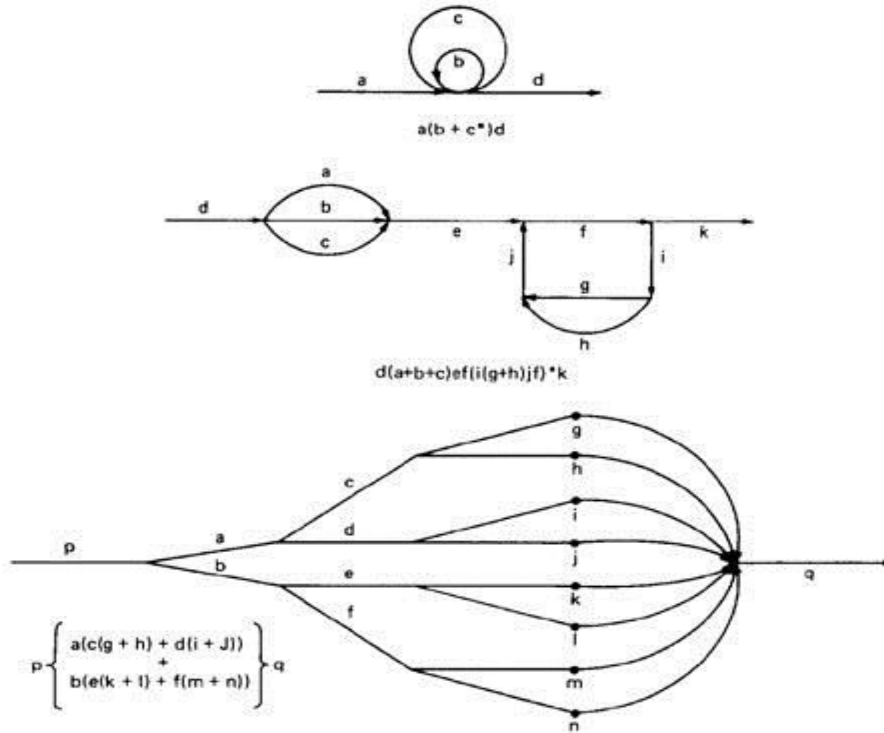


Figure 5.6: Some graphs and their path expressions.

APPLICATIONS - CO2

- **APPLICATIONS:**
  - The purpose of the node removal algorithm is to present one very generalized concept- the path expression and way of getting it.
  - Every application follows this common pattern:
    1. Convert the program or graph into a path expression.
    2. Identify a property of interest and derive an appropriate set of "arithmetic" rules that characterizes the property.
    3. Replace the link names by the link weights for the property of interest. The path expression has now been converted to an expression in some algebra, such as ordinary algebra, regular expressions, or boolean algebra. This algebraic expression summarizes the property of interest over the set of all paths.

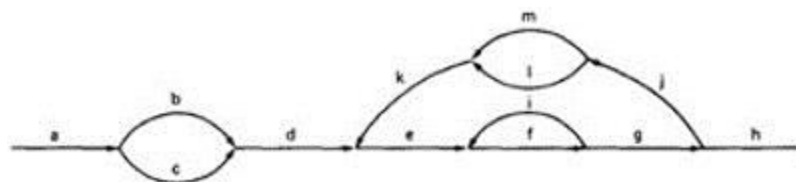
4. Simplify or evaluate the resulting "algebraic" expression to answer the question you asked.

- **HOW MANY PATHS IN A FLOWGRAPH ?**
  - The question is not simple. Here are some ways you could ask it:
    1. What is the maximum number of different paths possible?
    2. What is the fewest number of paths possible?
    3. How many different paths are there really?
    4. What is the average number of paths?
  - Determining the actual number of different paths is an inherently difficult problem because there could be unachievable paths resulting from correlated and dependent predicates.
  - If we know both of these numbers (maximum and minimum number of possible paths) we have a good idea of how complete our testing is.
  - Asking for "the average number of paths" is meaningless.
- **MAXIMUM PATH COUNT ARITHMETIC:**
  - Label each link with a link weight that corresponds to the number of paths that link represents.
  - Also mark each loop with the maximum number of times that loop can be taken. If the answer is infinite, you might as well stop the analysis because it is clear that the maximum number of paths will be infinite.
  - There are three cases of interest: parallel links, serial links, and loops.

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

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

    1. The following is a reasonably well-structured program.

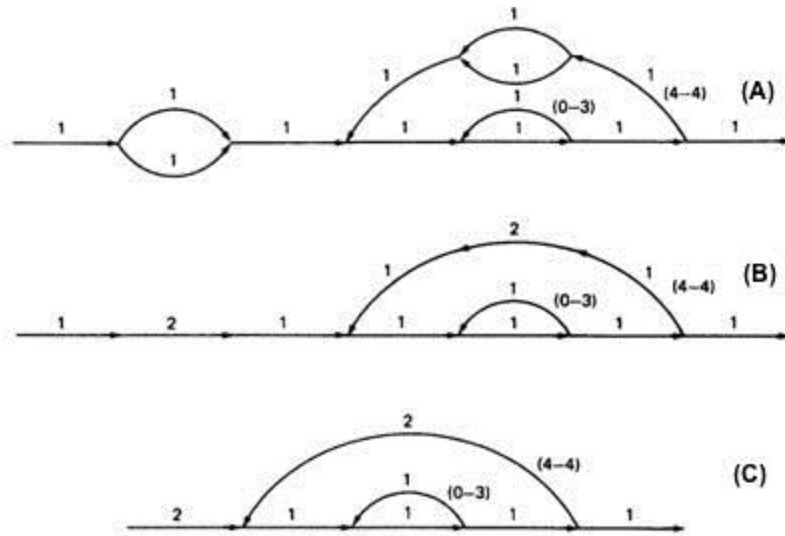


a(b + c)d{e(fi)*fgj(m + l)k}*e(fi)*fgh

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

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

2. **A:** The flow graph should be annotated by replacing the link name with the maximum of paths through that link (1) and also note the number of times for looping.
3. **B:** Combine the first pair of parallel loops outside the loop and also the pair in the outer loop.
4. **C:** Multiply the things out and remove nodes to clear the clutter.

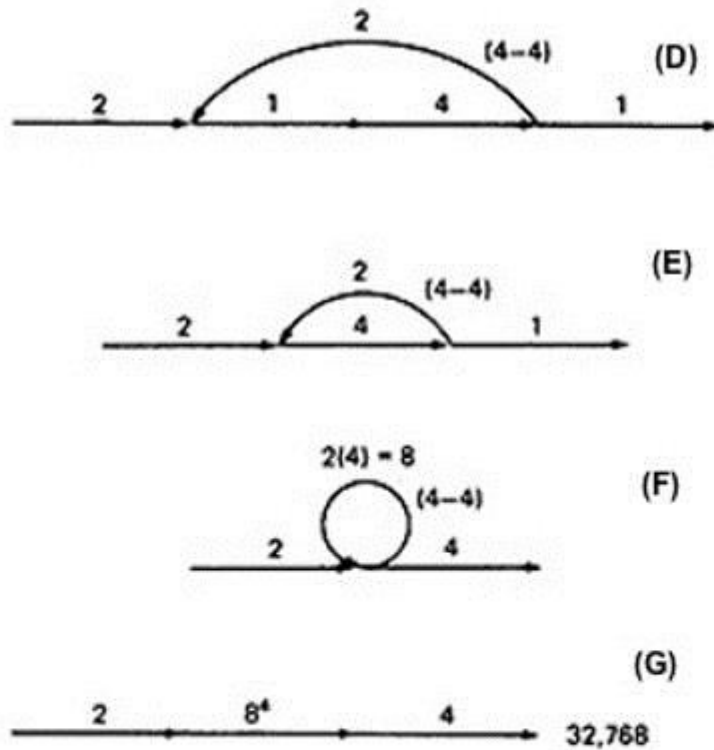


5. **For the Inner Loop:** **D:**Calculate the total weight of inner loop, which can execute a min. of 0 times and max. of 3 times. So, it inner loop can be evaluated as follows:

   $1^{\underline{3}} = 1^0 + 1^1 + 1^2 + 1^3 = 1 + 1 + 1 + 1 = 4$

6. **E:** Multiply the link weights inside the loop: 1 X 4 = 4
7. **F:** Evaluate the loop by multiplying the link wieghts: 2 X 4 = 8.
8. **G:** Simpifying the loop further results in the total maximum number of paths in the flowgraph:

   $2 \times 8^4 \times 2 = 32,768.$



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

  a(b+c)d{e(fi)*fgj(m+l)k}*e(fi)*fgh
  $= 1(1 + 1)1(1(1 \times 1)^3 1 \times 1 \times 1(1 + 1)1)^4 1(1 \times 1)^3 1 \times 1 \times 1$
  $= 2(1^3 1 \times (2))^4 1^3$
  $= 2(4 \times 2)^4 \times 4$
  $= 2 \times 8^4 \times 4 = 32{,}768$

- This is the same result we got graphically.
- Actually, the outer loop should be taken exactly four times. That doesn't mean it will be taken zero or four times. Consequently, there is a superfluous "4" on the outlink in the last step. Therefore the maximum number of different paths is 8192 rather than 32,768.
  - **STRUCTURED FLOWGRAPH:**
- Structured code can be defined in several different ways that do not involve ad-hoc rules such as not using GOTOs.
- A structured flowgraph is one that can be reduced to a single link by successive application of the transformations of Figure 5.7.

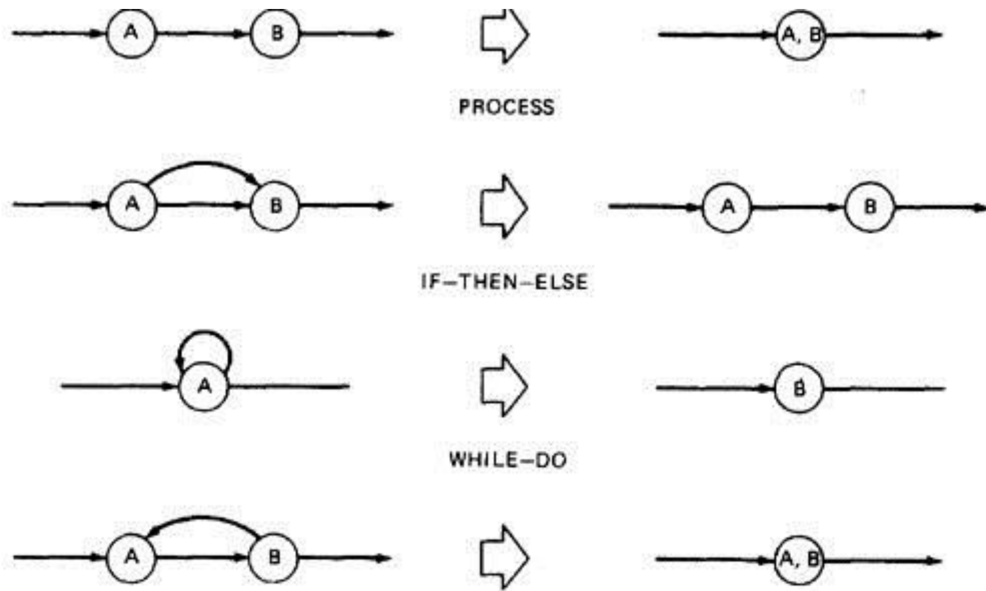


Figure 5.7: Structured Flowgraph Transformations.

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

    1. Jumping into loops
    2. Jumping out of loops
    3. Branching into decisions
    4. Branching out of decisions

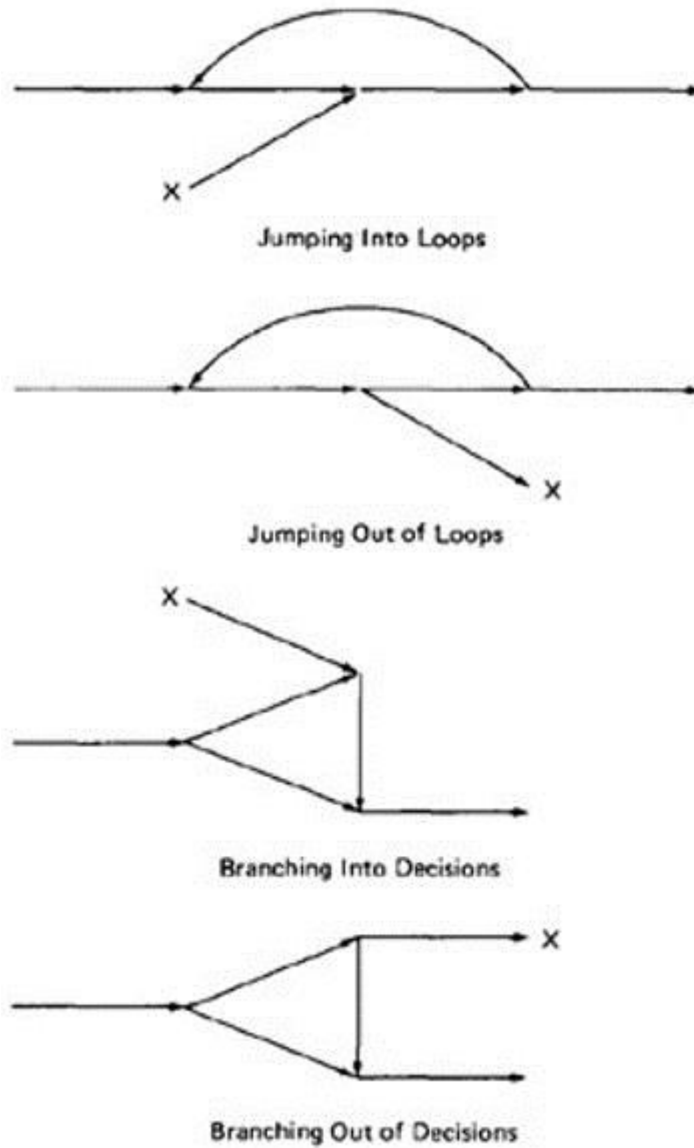


Figure 5.8: Un-structured sub-graphs.

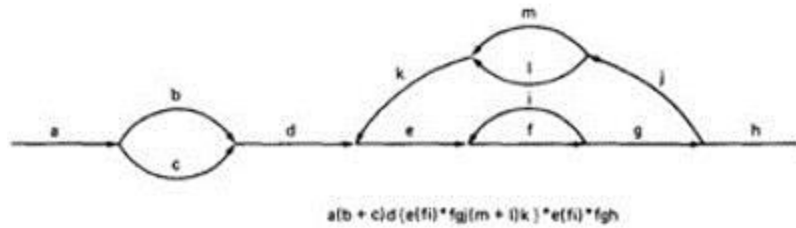
2. **LOWER PATH COUNT ARITHMETIC:**

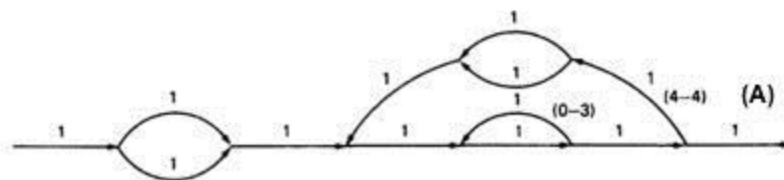
- A lower bound on the number of paths in a routine can be approximated for structured flow graphs.
- The arithmetic is as follows:

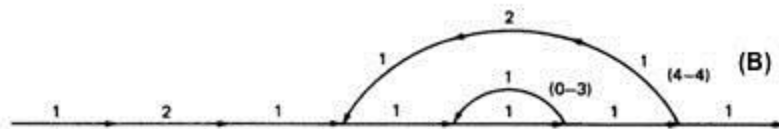
Case	Path expression	Weight expression
Parallels	A+B	W_A+W_B
Series	AB	$\max(W_A,W_B)$
Loop	$A^{\underline{n}}$	$1, W_1$

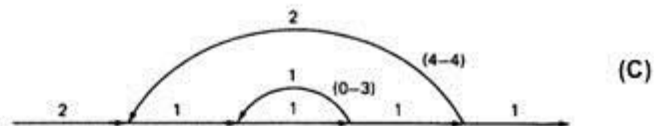
- The values of the weights are the number of members in a set of paths.
- **EXAMPLE:**

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

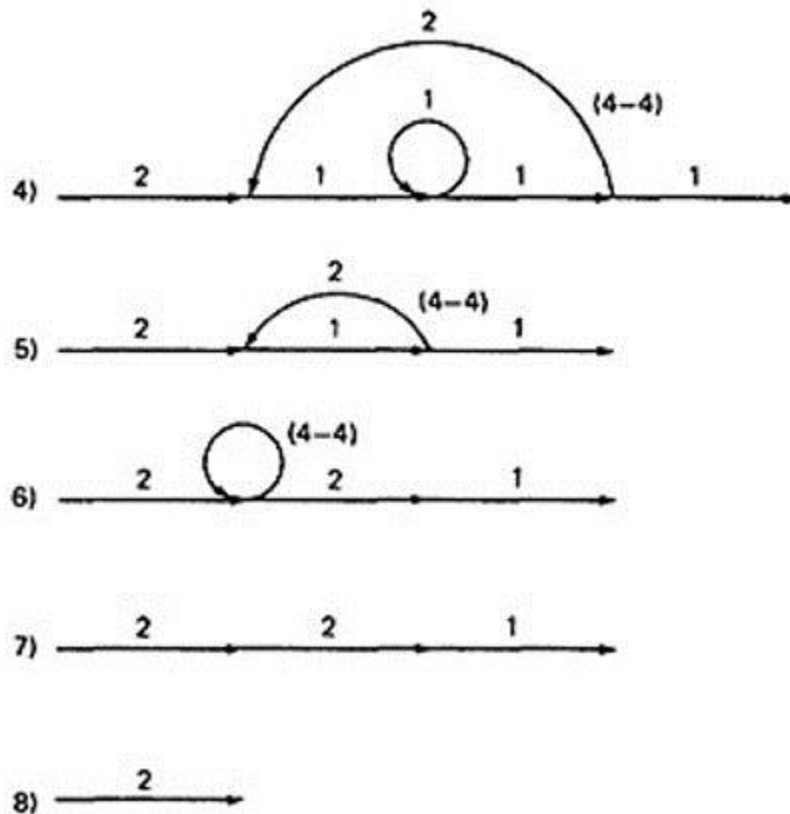








2. From Step 4, the it would be different from the previous example:



3. If you observe the original graph, it takes at least two paths to cover and that it can be done in two paths.
4. If you have fewer paths in your test plan than this minimum you probably haven't covered. It's another check.

- **CALCULATING THE PROBABILITY:**

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

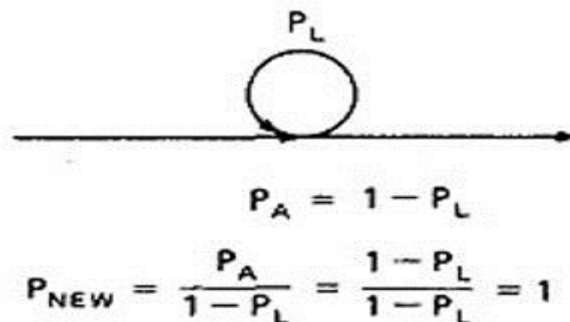
  What is the probability of being at a certain point in a routine?

  This question can be answered under suitable assumptions, primarily that all probabilities involved are independent, which is to say that all decisions are independent and uncorrelated.
- We use the same algorithm as before : node-by-node removal of uninteresting nodes.
- **Weights, Notations and Arithmetic:**
  1. Probabilities can come into the act only at decisions (including decisions associated with loops).

2. Annotate each outlink with a weight equal to the probability of going in that direction.
3. Evidently, the sum of the outlink probabilities must equal 1
4. For a simple loop, if the loop will be taken a mean of N times, the looping probability is N/(N + 1) and the probability of not looping is 1/(N + 1).
5. A link that is not part of a decision node has a probability of 1.
6. The arithmetic rules are those of ordinary arithmetic.

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

7. In this table, in case of a loop, P_A is the probability of the link leaving the loop and P_L is the probability of looping.
8. The rules are those of ordinary probability theory.
    1. If you can do something either from column A with a probability of P_A or from column B with a probability P_B, then the probability that you do either is $P_A + P_B$.
    2. For the series case, if you must do both things, and their probabilities are independent (as assumed), then the probability that you do both is the product of their probabilities.
9. For example, a loop node has a looping probability of P_L and a probability of not looping of P_A, which is obviously equal to I - PL.



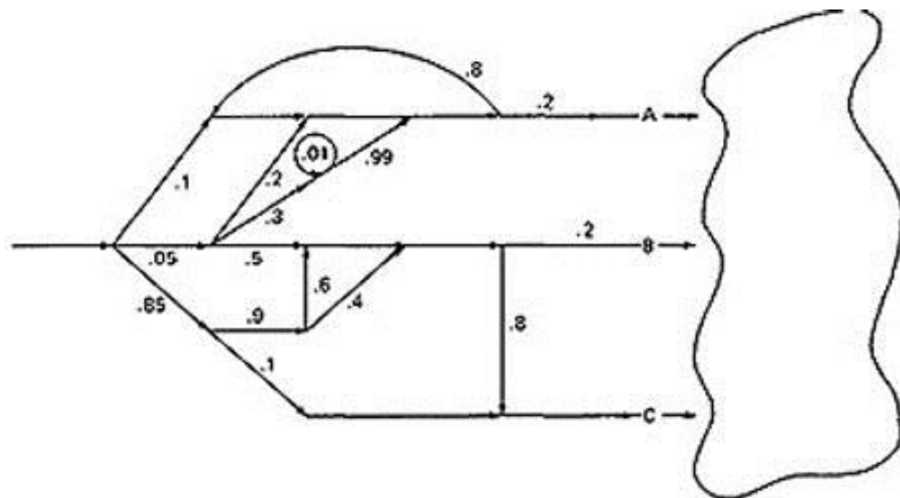
10. Following the above rule, all we've done is replace the outgoing probability with 1 - so why the complicated rule? After a few steps in which you've removed nodes, combined parallel terms, removed loops and the like, you might find something like this:

because $P_L + P_A + P_B + P_C = 1$, $1 - P_L = P_A + P_B + P_C$, and

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

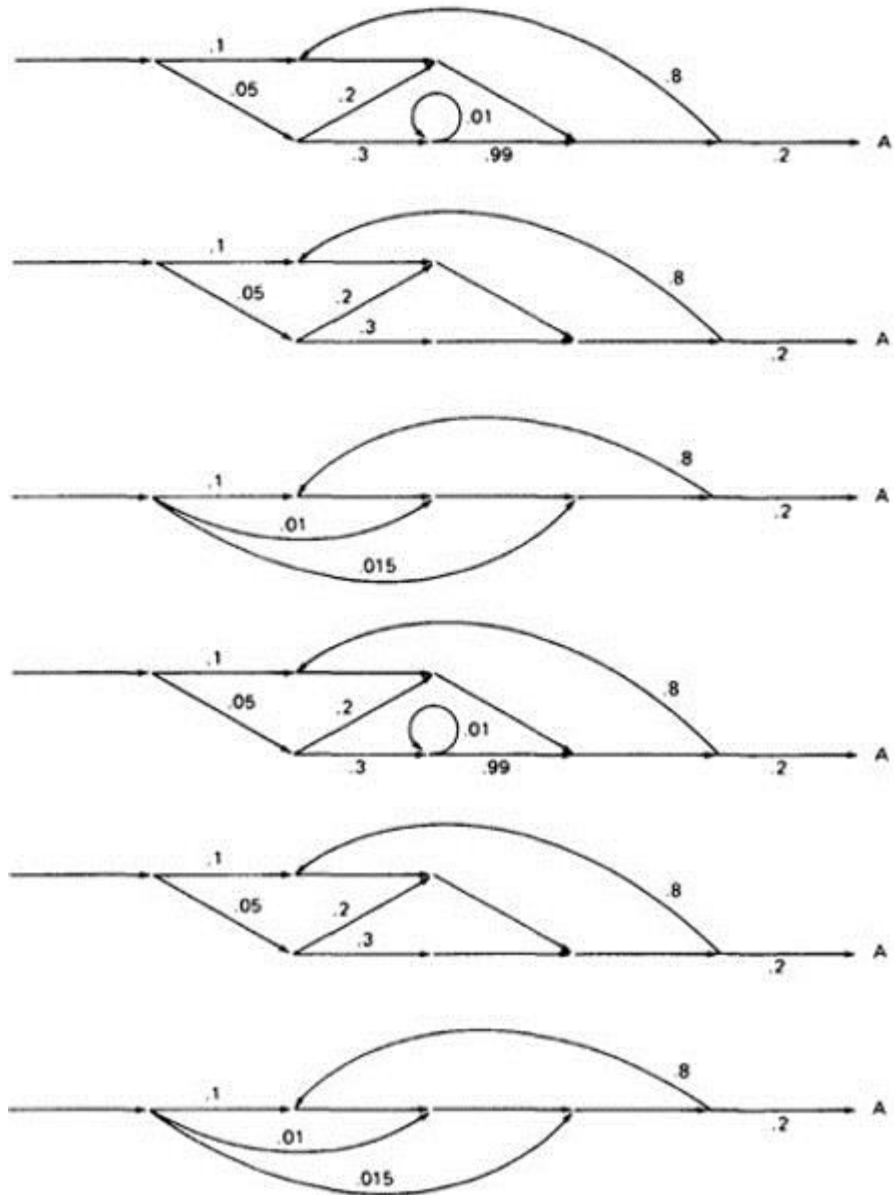
which is what we've postulated for any decision. In other words, division by $1 - P_L$ renormalizes the outlink probabilities so that their sum equals unity after the loop is removed.

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

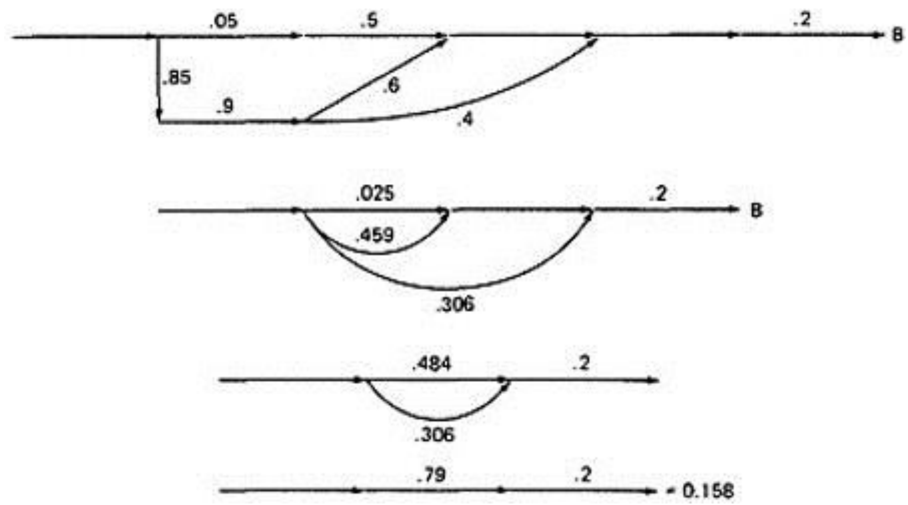


2. Let us do this in three parts, starting with case A. Note that the sum of the probabilities at each decision node is equal to 1. Start by throwing away anything that isn't on the way to case A, and then apply the reduction procedure. To avoid clutter, we usually leave out probabilities equal to 1.

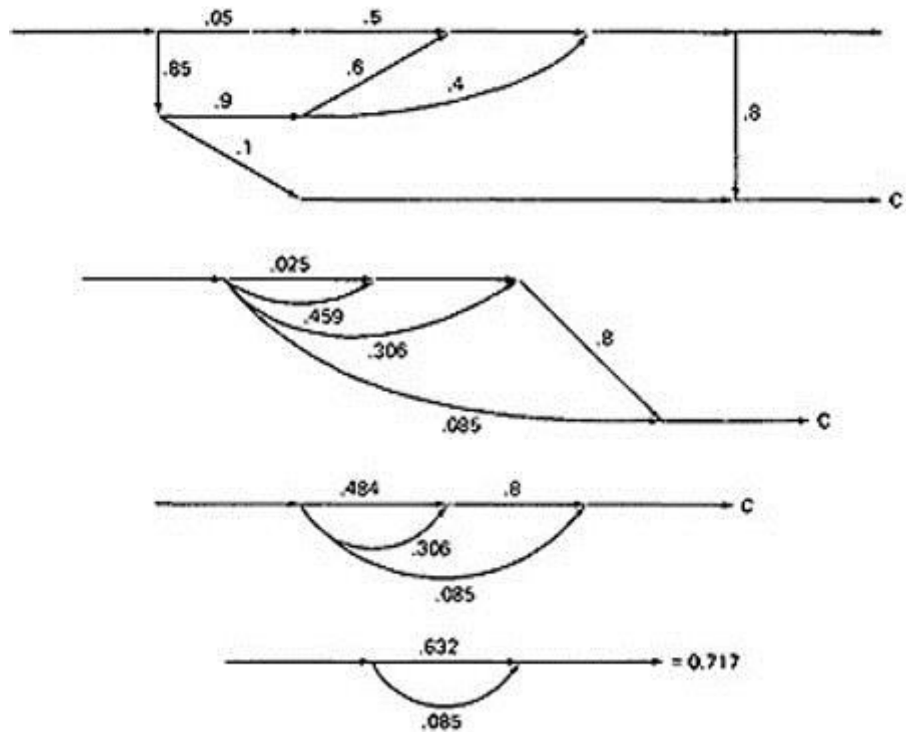
   CASE A:



3. Case B is simpler:



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



5. This checks. It's a good idea when doing this sort of thing to calculate all the probabilities and to verify that the sum of the routine's exit probabilities does equal 1.
6. If it doesn't, then you've made calculation error or, more likely, you've left out some branching probability.
7. How about path probabilities? That's easy. Just trace the path of interest and multiply the probabilities as you go.
8. Alternatively, write down the path name and do the indicated arithmetic operation.
9. Say that a path consisted of links a, b, c, d, e, and the associated probabilities were .2, .5, 1., .01, and I respectively. Path *abcbcbcdeabddea* would have a probability of 5×10^{-10}.
10. Long paths are usually improbable.

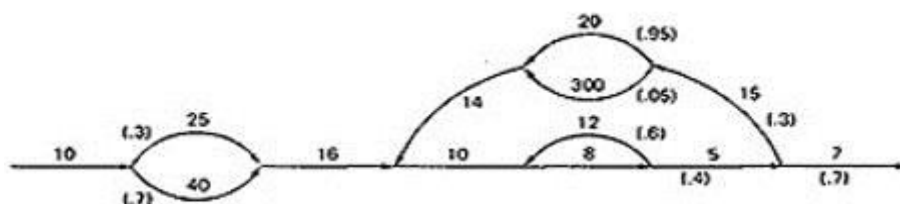
- **MEAN PROCESSING TIME OF A ROUTINE:**

- Given the execution time of all statements or instructions for every link in a flowgraph and the probability for each direction for all decisions are to find the mean processing time for the routine as a whole.
- The model has two weights associated with every link: the processing time for that link, denoted by **T**, and the probability of that link **P**.
- The arithmetic rules for calculating the mean time:

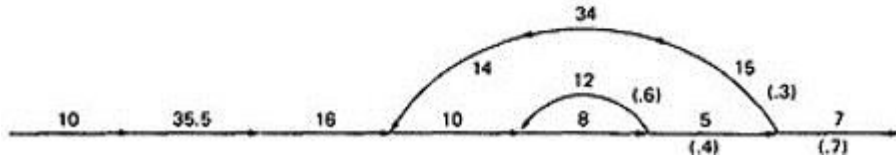
Case	Path expression	Weight expression
Parallel	A+B	$T_{A+B}=(P_A T_A+P_B T_B)/(P_A+P_B)$ $P_{A+B}= P_A+P_B$
Series	AB	$T_{AB}=T_A+T_B$ $P_{AB} =P_A P_B$
Loop	A^n	$T_A= T_A+ T_L P_L/(1-P_L)$ $P_A =P_A/(1-P_L)$

- **EXAMPLE:**

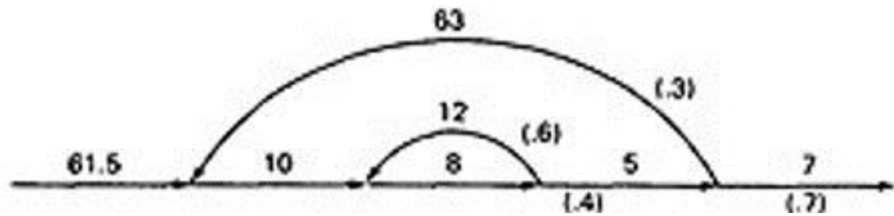
1. Start with the original flow graph annotated with probabilities and processing time.



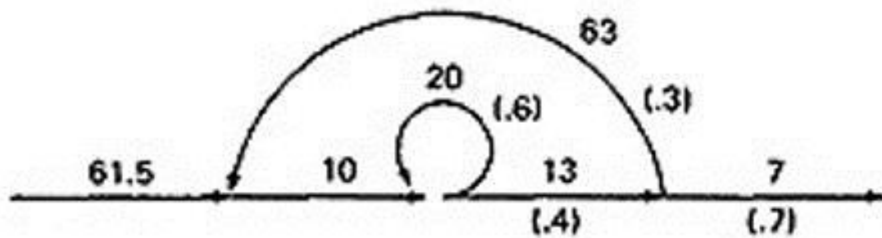
2. Combine the parallel links of the outer loop. The result is just the mean of the processing times for the links because there aren't any other links leaving the first node. Also combine the pair of links at the beginning of the flowgraph..



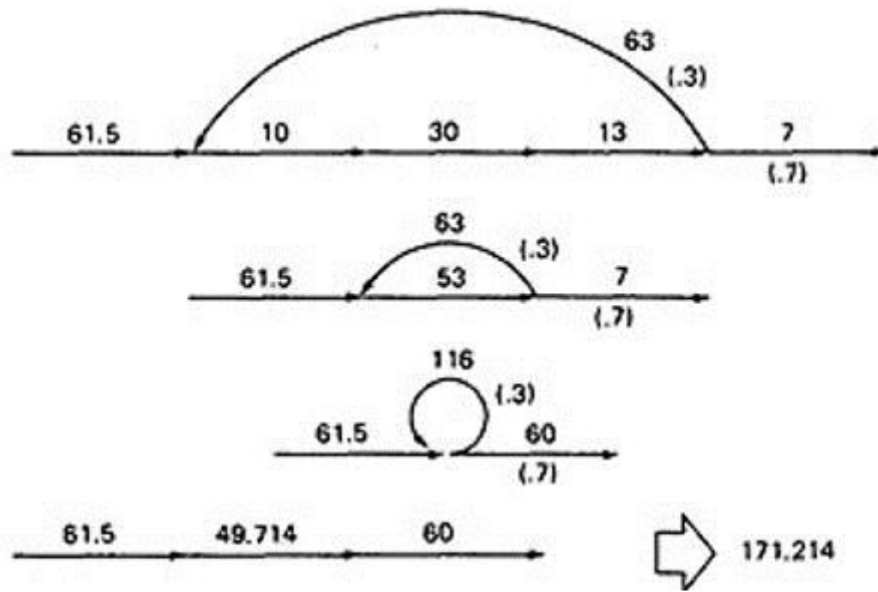
3. Combine as many serial links as you can.



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



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



2. **PUSH/POP, GET/RETURN:**

- This model can be used to answer several different questions that can turn up in debugging.
- It can also help decide which test cases to design.
- The question is:

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

- Here are some other examples of complementary operations to which this model applies:
- GET/RETURN a resource block.
- OPEN/CLOSE a file.
  START/STOP a device or process.

- **EXAMPLE 1 (PUSH / POP):**

1. Here is the Push/Pop Arithmetic:

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

2. The numeral 1 is used to indicate that nothing of interest (neither PUSH nor POP) occurs on a given link.
3. "H" denotes PUSH and "P" denotes POP. The operations are commutative, associative, and distributive.

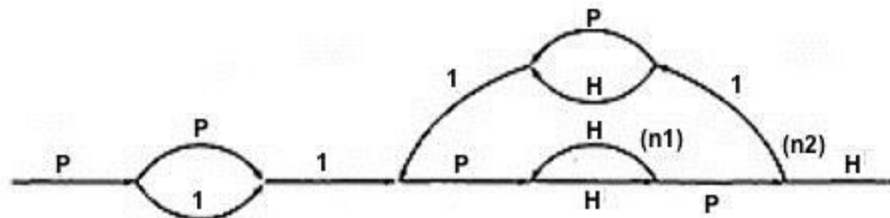
PUSH/POP MULTIPLICATION TABLE

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

PUSH/POP ADDITION TABLE

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

4. Consider the following flowgraph:



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

5. Simplifying by using the arithmetic tables,
6. $= (P^2 + P)\{P(HH)^{n1}(P + H)\}^{n1}(HH)^{n1}$
7. $= (P^2 + P)\{H^{2n1}(P^2 + 1)\}^{n2}H^{2n1}$
8. Below Table 5.9 shows several combinations of values for the two looping terms - M1 is the number of times the inner loop will be taken and M2 the number of times the outer loop will be taken.

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

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

9. These expressions state that the stack will be popped only if the inner loop is not taken.
10. The stack will be left alone only if the inner loop is iterated once, but it may also be pushed.
11. For all other values of the inner loop, the stack will only be pushed.

- **EXAMPLE 2 (GET / RETURN):**

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

Multiplication Table

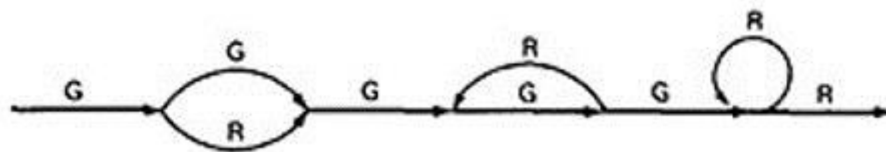
×	G	R	1
G	G^2	1	G
R	1	R^2	R
1	G	R	1

Addition Table

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

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

3. Consider the following flowgraph:



4. $G(G + R)G(GR)^*GGR^*R$
$= G(G + R)G^3R^*R$
$= (G + R)G^3R^*$
$= (G^4 + G^2)R^*$
5. This expression specifies the conditions under which the resources will be balanced on leaving the routine.
6. If the upper branch is taken at the first decision, the second loop must be taken four times.

7. If the lower branch is taken at the first decision, the second loop must be taken twice.
8. For any other values, the routine will not balance. Therefore, the first loop does not have to be instrumented to verify this behavior because its impact should be nil.

LIMITATIONS AND SOLUTIONS:

- The main limitation to these applications is the problem of unachievable paths.
- The node-by-node reduction procedure, and most graph-theory-based algorithms work well when all paths are possible, but may provide misleading results when some paths are unachievable.
- The approach to handling unachievable paths (for any application) is to partition the graph into subgraphs so that all paths in each of the subgraphs are achievable.
- The resulting subgraphs may overlap, because one path may be common to several different subgraphs.
- Each predicate's truth-functional value potentially splits the graph into two subgraphs. For n predicates, there could be as many as 2^n subgraphs.
-

REGULAR EXPRESSIONS AND FLOW ANOMALY DETECTION - CO2

THE PROBLEM:

- The generic flow-anomaly detection problem (note: not just data-flow anomalies, but any flow anomaly) is that of looking for a specific sequence of options considering all possible paths through a routine.
- Let the operations be SET and RESET, denoted by s and r respectively, and we want to know if there is a SET followed immediately a SET or a RESET followed immediately by a RESET (an *ss* or an *rr* sequence).
- Some more application examples:
    1. A file can be opened (o), closed (c), read (r), or written (w). If the file is read or written to after it's been closed, the sequence is nonsensical. Therefore, *cr* and *cw* are anomalous. Similarly, if the file is read before it's been written, just after opening, we may have a bug. Therefore, *or* is also anomalous. Furthermore, *oo* and *cc*, though not actual bugs, are a waste of time and therefore should also be examined.
    2. A tape transport can do a rewind (d), fast-forward (f), read (r), write (w), stop (p), and skip (k). There are rules concerning the use of the transport; for example, you cannot go from rewind to fast-forward without an intervening stop or from rewind or fast-forward to read or write without an intervening stop. The following sequences are anomalous: *df*, *dr*, *dw*, *fd*, and *fr*. Does the flowgraph lead to anomalous sequences on any path? If so, what sequences and under what circumstances?
    3. The data-flow anomalies discussed in Unit 4 requires us to detect the *dd*, *dk*, *kk*, and *ku* sequences. Are there paths with anomalous data flows?
    4.

- **THE METHOD:**
  - Annotate each link in the graph with the appropriate operator or the null operator 1.
  - Simplify things to the extent possible, using the fact that a + a = a and 12 = 1.
  - You now have a regular expression that denotes all the possible sequences of operators in that graph. You can now examine that regular expression for the sequences of interest.
  - **EXAMPLE:** Let A, B, C, be nonempty sets of character sequences whose smallest string is at least one character long. Let T be a two-character string of characters. Then if T is a substring of (i.e., if T appears within) AB^nC, then T will appear in AB^2C. (**HUANG's Theorem**)
  - As an example, let

    A= *pp*
    B= *srr*
    C= *rp*
    T= *ss*

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

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

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

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

    Multiplying out the expression and simplifying shows that there is no p^4 sequence.
  - Incidentally, the above observation is an informal proof of the wisdom of looping twice discussed in Unit 2. Because data-flow anomalies are represented by two-character sequences, it follows the above theorem that looping twice is what you need to do to find such anomalies.
- **LIMITATIONS:**
  - Huang's theorem can be easily generalized to cover sequences of greater length than two characters. Beyond three characters, though, things get complex and this method has probably reached its utilitarian limit for manual application.
  - There are some nice theorems for finding sequences that occur at the beginnings and ends of strings but no nice algorithms for finding strings buried in an expression.
  - Static flow analysis methods can't determine whether a path is or is not achievable. Unless the flow analysis includes symbolic execution or similar techniques, the impact of unachievable paths will not be included in the analysis.

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

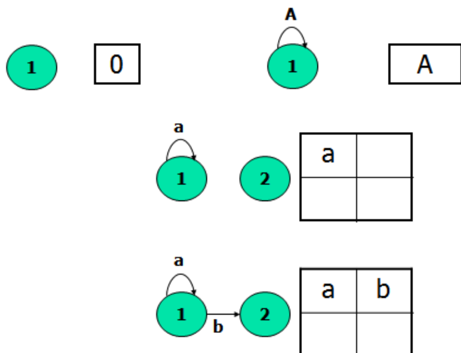
CVR COLLEGE OF ENGINEERING

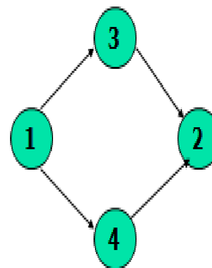
An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 1
Unit - 5
Year and Semester: IV yr & I Sem
Subject**: Software Testing Methodologies**
Branch: **CSE**
Faculty: **Revathi Lavanya Baggam,** Assistant Professor (CSE)

MATRIX OF A GRAPH - CO4

- A graph matrix is a square array with one row and one column for every node in the graph.
- Each row-column combination corresponds to a relation between the node corresponding to the row and the node corresponding to the column.
- The relation for example, could be as simple as the link name, if there is a link between the nodes.
- Some of the things to be observed:
- The size of the matrix equals the number of nodes.
- There is a place to put every possible direct connection or link between any and any other node.
- The entry at a row and column intersection is the link weight of the link that connects the two nodes in that direction.
- A connection from node i to j does not imply a connection from node j to node i.
- If there are several links between two nodes, then the entry is a sum; the "+" sign denotes parallel links as usual.





		a	c
	b		
	d		

RELATIONS - CO4

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

For example,

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

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

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

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

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

Transitive:

A relation is transitive if aRb and bRc implies aRc.

Most relations used in testing are transitive.

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

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

REFLEXIVE

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

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

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

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

SYMMETRIC

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

A symmetric relation mean that if there is a link from a to b then there is also a link from b to a.

A graph whose relations are not symmetric are called directed graph.

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

The matrix of an undirected graph is symmetric ($a_{ij}=a_{ji}$) for all i,j)

ANTI SYMMETRIC

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

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

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

EQUIVALLENCE

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

Equality is the most familiar example of an equivalence relation.

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

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

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

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

PARTIAL ORDERING

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

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

POWER OF A MATRIX - CO4

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

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

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

The third power represents all path segments three links long.

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

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

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

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

NODE REDUCTION ALGORITHM - CO4

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

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

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

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

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

Combine the parallel terms and simplify as you can.

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

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

SYSTEM AND ACCEPTANCE TESTING

Functional System Testing - CO3

Functional Testing is defined as a type of testing which verifies that each **function** of the software application operates in conformance with the requirement specification. This testing mainly involves black box testing and it is not concerned about the source code of the application.

Each and every functionality of the system is tested by providing appropriate input, verifying the output and comparing the actual results with the expected results.
This testing involves checking of User Interface, APIs, Database, security, client/ server applications and functionality of the Application Under Test. The testing can be done either manually or using automation

The prime objective of Functional testing is checking the functionalities of the software system. It mainly concentrates on -

- **Mainline functions**: Testing the main functions of an application
- **Basic Usability**: It involves basic usability testing of the system. It checks whether a user can freely navigate through the screens without any difficulties.
- **Accessibility**: Checks the accessibility of the system for the user
- **Error Conditions**: Usage of testing techniques to check for error conditions. It checks whether suitable error messages are displayed.

How to perform Functional Testing: Complete Process

In order to functionally test an application, the following steps must be observed.

- Understand the Software Engineering Requirements
- Identify test input (test data)
- Compute the expected outcomes with the selected test input values
- Execute test cases
- Comparison of actual and computed expected result

Non Functional System Testing - CO3

Non-functional testing is the testing of a software application or system for its non-functional requirements: the way a system operates, rather than specific behaviours of that system. This is in contrast to functional testing, which tests against functional requirements that describe the functions of a system and its components. The names of many non-functional tests are often used interchangeably because of the overlap in scope between various non-functional requirements. For example, software performance is a broad term that includes many specific requirements like reliability and scalability.

Non-functional testing includes:

- Baseline testing
- Compliance testing
- Documentation testing
- Endurance testing
- Load testing
- Localization testing and Internationalization testing
- Performance testing
- Recovery testing
- Resilience testing
- Security testing
- Scalability testing

- Stress testing
- Usability testing
- Volume testing

Objectives of Non-functional testing

- Non-functional testing should increase usability, efficiency, maintainability, and portability of the product.
- Helps to reduce production risk and cost associated with non-functional aspects of the product.
- Optimize the way product is installed, setup, executes, managed and monitored.
- Collect and produce measurements, and metrics for internal research and development.
- Improve and enhance knowledge of the product behavior and technologies in use.

Characteristics of Non-functional testing

- Non-functional testing should be measurable, so there is no place for subjective characterization like good, better, best, etc.
- Exact numbers are unlikely to be known at the start of the requirement process
- Important to prioritize the requirements
- Ensure that quality attributes are identified correctly in Software Engineering.

ACCEPTANCE TESTING - CO5

ACCEPTANCE TESTING is a level of software testing 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.

<u>Definition by ISTQB</u>

- **Acceptance testing:** Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.

Analogy

During the process of manufacturing a ballpoint pen, the cap, the body, the tail and clip, the ink cartridge and the ballpoint are produced separately and unit tested separately. When two or more units are ready, they are assembled and Integration Testing is performed. When the complete pen is integrated, System Testing is performed. Once System Testing is complete, Acceptance Testing is performed so as to confirm that the ballpoint pen is ready to be made available to the end-users.

Method

Usually, Black Box Testing method is used in Acceptance Testing. Testing does not normally follow a strict procedure and is not scripted but is rather ad-hoc.

Tasks

- Acceptance Test Plan
  - Prepare
  - Review
  - Rework
  - Baseline
- Acceptance Test Cases/Checklist
  - Prepare
  - Review
  - Rework
  - Baseline
- Acceptance Test
  - Perform

When is it performed?

Acceptance Testing is the fourth and last level of software testing performed after System Testing and before making the system available for actual use.

Who performs it?

- Internal Acceptance Testing (Also known as Alpha Testing) is performed by members of the organization that developed the software but who are not directly involved in the project (Development or Testing). Usually, it is the members of Product Management, Sales and/or Customer Support.

- External Acceptance Testing is performed by people who are not employees of the organization that developed the software.
  - Customer Acceptance Testing is performed by the customers of the organization that developed the software. They are the ones who asked the organization to develop the software. [This is in the case of the software not being owned by the organization that developed it.]
  - User Acceptance Testing (Also known as Beta Testing) is performed by the end users of the software. They can be the customers themselves or the customers' customers.

TESTING OBJECT ORIENTED SYSTEMS - CO5

The shift from traditional to object-oriented environment involves looking at and reconsidering old strategies and methods for testing the software. The traditional programming consists of procedures operating on data, while the object-oriented paradigm focuses on objects that are instances of classes. In object-oriented (OO) paradigm, software engineers identify and specify the objects and services provided by each object. In addition, interaction of any two objects and constraints on each identified object are also determined. The main advantages of OO paradigm include increased reusability, reliability, interoperability, and extendibility.

With the adoption of OO paradigm, almost all the phases of software development have changed in their approach, environments, and tools. Though OO paradigm helps make the designing and development of software easier, it may pose new kind of problems. Thus, testing of software developed using OO paradigm has to deal with the new problems also. Note that object-oriented testing can be used to test the object-oriented software as well as conventional software.

OO program should be tested at different levels to uncover all the errors. At the algorithmic level, each module (or method) of every class in the program should be tested in isolation. For this, white-box testing can be applied easily. As classes form the main unit of object-oriented program, testing of classes is the main concern while testing an OO program. At the class level, every class should be tested as an individual entity. At this level, programmers who are involved in the development of class conduct the testing. Test cases can be drawn from requirements specifications, models, and the language used. In addition, structural testing methods such as boundary value analysis are extremely used. After performing the testing at class level, cluster level testing should be performed. As classes are collaborated (or integrated) to form a small subsystem (also known as cluster), testing each cluster individually is necessary. At this level, focus is on testing the components that execute concurrently as well as on the interclass interaction. Hence, testing at this level may be viewed as integration testing where units to be integrated are classes. Once all the clusters in the system are tested, system level testing begins. At this level, interaction among clusters is tested.

Usually, there is a misconception that if individual classes are well designed and have proved to work in isolation, then there is no need to test the interactions between two or more classes when they are integrated. However, this is not true because sometimes there can be errors, which can be detected only through integration of classes. Also, it is possible that if a class does not contain a bug, it may still be used in a wrong way by another class, leading to system failure.

Developing Test Cases in Object-oriented Testing

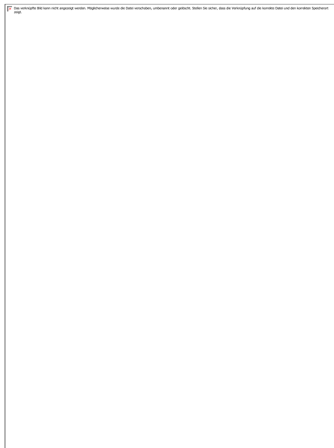
The methods used to design test cases in OO testing are based on the conventional methods. However, these test cases should encompass special features so that they can be used in the object-oriented environment. The points that should be noted while developing test cases in an object-oriented environment are listed below.

1.It should be explicitly specified with each test case which class it should test.
2.Purpose of each test case should be mentioned.

3.External conditions that should exist while conducting a test should be clearly stated with each test case.
4.All the states of object that is to be tested should be specified.
5.Instructions to understand and conduct the test cases should be provided with each test case.

Object-oriented Testing Methods

As many organizations are currently using or targeting to switch to the OO paradigm, the importance of OO software testing is increasing. The methods used for performing object-oriented testing are discussed in this section.



State-based testing is used to verify whether the methods (a procedure that is executed by an object) of a class are interacting properly with each other. This testing seeks to exercise the transitions among the states of objects based upon the identified inputs.

For this testing, finite-state machine (FSM) or state-transition diagram representing the possible states of the object and how state transition occurs is built. In addition, state-based testing generates test cases, which check whether the method is able to change the state of object as expected. If any method of the class does not change the object state as expected, the method is said to contain errors.

To perform state-based testing, a number of steps are followed, which are listed below.

1.Derive a new class from an existing class with some additional features, which are used to examine and set the state of the object.
2.Next, the test driver is written. This test driver contains a main program to create an object, send messages to set the state of the object, send messages to invoke methods of the class that is being tested and send messages to check the final state of the object.
3.Finally, stubs are written. These stubs call the untested methods.

Fault-based Testing

Fault-based testing is used to determine or uncover a set of plausible faults. In other words, the focus of tester in this testing is to detect the presence of possible faults. Fault-based testing starts by examining the analysis and design models of OO software as these models may provide an idea of problems in the

implementation of software. With the knowledge of system under test and experience in the application domain, tester designs test cases where each test case targets to uncover some particular faults.

The effectiveness of this testing depends highly on tester experience in application domain and the system under test. This is because if he fails to perceive real faults in the system to be plausible, testing may leave many faults undetected. However, examining analysis and design models may enable tester to detect large number of errors with less effort. As testing only proves the existence and not the absence of errors, this testing approach is considered to be an effective method and hence is often used when security or safety of a system is to be tested.

Integration testing applied for OO software targets to uncover the possible faults in both operation calls and various types of messages (like a message sent to invoke an object). These faults may be unexpected outputs, incorrect messages or operations, and incorrect invocation. The faults can be recognized by determining the behavior of all operations performed to invoke the methods of a class.

Scenario-based Testing

Scenario-based testing is used to detect errors that are caused due to incorrect specifications and improper interactions among various segments of the software. Incorrect interactions often lead to incorrect outputs that can cause malfunctioning of some segments of the software. The use of scenarios in testing is a common way of describing how a user might accomplish a task or achieve a goal within a specific context or environment. Note that these scenarios are more context- and user specific instead of being product-specific. Generally, the structure of a scenario includes the following points.

1.A condition under which the scenario runs.
2.A goal to achieve, which can also be a name of the scenario.
3.A set of steps of actions.
4.An end condition at which the goal is achieved.
5.A possible set of extensions written as scenario fragments.
Scenario- based testing combines all the classes that support a use-case (scenarios are subset of use-cases) and executes a test case to test them. Execution of all the test cases ensures that all methods in all the classes are executed at least once during testing. However, testing all the objects (present in the classes combined together) collectively is difficult. Thus, rather than testing all objects collectively, they are tested using either top-down or bottom-up integration approach.

This testing is considered to be the most effective method as scenarios can be organized in such a manner that the most likely scenarios are tested first with unusual or exceptional scenarios considered later in the testing process. This satisfies a fundamental principle of testing that most testing effort should be devoted to those paths of the system that are mostly used.

Challenges in Testing Object-oriented Programs

Traditional testing methods are not directly applicable to OO programs as they involve OO concepts including encapsulation, inheritance, and polymorphism. These concepts lead to issues, which are yet to be resolved. Some of these issues are listed below.

1.Encapsulation of attributes and methods in class may create obstacles while testing. As methods are invoked through the object of corresponding class, testing cannot be accomplished without object. In

addition, the state of object at the time of invocation of method affects its behavior. Hence, testing depends not only on the object but on the state of object also, which is very difficult to acquire.
2.Inheritance and polymorphism also introduce problems that are not found in traditional software. Test cases designed for base class are not applicable to derived class always (especially, when derived class is used in different context). Thus, most testing methods require some kind of adaptation in order to function properly in an OO environment.

DIFFERENCES IN OO TESTING - CO5

What Is Different about Testing Object-Oriented Software?

Object-oriented programming features in programming languages obviously impact some aspects of testing. Features such as class inheritance and interfaces support polymorphism in which code manipulates objects without their exact class being known. Testers must ensure the code works no matter what the exact class of such objects might be. Language features that support and enforce data hiding can complicate testing because operations must sometimes be added to a class interface just to support testing. On the other hand, the availability of these features can contribute to better and reusable testing software.
Not only do changes in programming languages affect testing, but so do changes in the development process and changes in the focus of analysis and design. Many object-oriented software-testing activities have counterparts in traditional processes. We still have a use for unit testing although the meaning of *unit* has changed. We still do integration testing to make sure various subsystems can work correctly in concert. We still need system testing to verify that software meets requirements. We still do regression testing to make sure the latest round of changes to the software hasn't adversely affected what it could do before.
The differences between "old" and "new" ways of developing and testing software are much deeper than a focus on objects instead of on functions that transform inputs to outputs. The most significant difference is in the way object-oriented software is designed as a set of objects that essentially model a problem and then collaborate to effect a solution. Underlying this approach is the concept that while a solution to a problem might need to change over time, the structure and components of the problem itself do not change as much or as frequently. Consequently, a program whose design is structured from the problem (and not on an immediately required solution) will be more adaptable to changes later. A programmer familiar with the problem and its components can recognize them in the software, thereby making the program more maintainable. Furthermore, because components are derived from the problem, they can often be reused in the development of other programs to solve similar or related problems, thereby improving the reusability of software components.
A big benefit of this approach to design is that analysis models map straightforwardly to design models that, in turn, map to code. Thus, we can start testing during analysis and refine the tests done in analysis to tests for design. Tests for design, in turn, can be refined to tests of implementation. This means that a testing process can be interwoven with the development process. We see three significant advantages to testing analysis and design models:

1. Test cases can be identified earlier in the process, even as requirements are being determined. Early testing helps analysts and designers to better understand and express requirements and to ensure that specified requirements are "testable."
2. Bugs can be detected early in the development process, saving time, money, and effort. It is widely acknowledged that the sooner problems are detected, the easier and cheaper they are to fix.
3. Test cases can be reviewed for correctness early in a project. The correctness of test cases—in particular, system test cases—is always an issue. If test cases are identified early and applied to models early in a project, then any misunderstandings of requirements on the part of the testers can be corrected early. In other words, model testing helps to ensure that testers and developers have a consistent understanding of system requirements.

Although testing models is very beneficial, it is important to not let testing them become the sole focus of testing efforts. Code testing is still an important part of the process.

Another difference between traditional projects and projects using object-oriented technologies concerns objectives for software. Consider, for example, that an important new goal in many companies is to produce reusable software, extensible designs, or even object-oriented frameworks that represent reusable designs. Testing can (and should) be done to uncover failures in meeting these objectives. Traditional testing approaches and techniques do not address such objectives.