**Rajeev Gandhi Memorial College of Engineering and Technology (Autonomous)**

**Nandyal**

**Department of CSE**

**IV B.Tech I SEM**

**Software Testing Methodologies and Tools (R15 Regulations)**

**Index**

**Rajeev Gandhi Memorial College of Engineering and Technology (Autonomous)**

**Nandyal**

**Department of CSE**

**IV-I B.Tech I$^{st}$ SEM (R15) Regulations**

**Software Testing Methodologies and Tools**

## Unit – I

***Objective of the Unit***

- Students know what the purpose of Testing
- Understand the differences between Tester and debugger, functional versus structure, builder versus buyer etc. understand the levels of testing and know the classification of bugs.

***Topic:*** **Purpose of Testing**

***Motivation (Why this topic is significant for the discussion?):***

- In the early period of 1970 the people thought that both the Testing and debugging are same so first know what are the different kinds bugs, these bugs are not possible to reduce with the help of debugging.

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

MYTH: Good programmers write code without bugs. (It's wrong!!!)

History says that even well written programs still have 1-3 bugs per hundred statements.

**Productivity and Quality in software:**

- In production of consumer goods and other products, every manufacturing stage is subjected to quality control and testing fromcomponent to final stage.
- If flaws are discovered at any stage, the product is either discarded or cycled back for rework and correction.
- Productivity is measured by the sum of the costs of the material, the rework, and the discarded components, and the cost of quality assurance and testing.
- There is a tradeoff between quality assurance costs and manufacturing costs: If sufficient time is not spent in quality assurance, the reject rate will be high and so will be the net cost. If inspection is good and all errors are caught as they occur, inspection costs will dominate, and again the net cost will suffer.
- Testing and Quality assurance costs for 'manufactured' items can be as low as 2% in consumer products or as high as 80% in products such as space-ships, nuclear reactors, and aircrafts, where failures threaten life. Whereas the manufacturing cost ofsoftware is trivial.
- The biggest part of software cost is the cost of bugs: the costof 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 indistinguishablebecause the cost of a software copy is trivial.

- Phases in a tester's mental life can be categorized into the following 5 phases:
    1. **Phase 0:** 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.
    2. **Phase 1:** the purpose of testing here is to show that software works. Highlighted during the late 1970s. This failed because the probability of showing that software works 'decreases' as testing increases. i.*e.* The more you test, the more likely you'll find a bug.
    3. **Phase 2:** the purpose of testing is to show that software doesn't work. This also failed because the software will never get released as you will find one bug or the other. Also, a bug corrected may also lead to another bug.
    4. **Phase 3:** the purpose of testing is not to prove anything butto 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.)
    5. **Phase 4:** Testability is the factor considered here. One reason is to reduce the labor of testing. Other reason is to check the testable and non-testable code. Testable code has fewer bugs than the code that's hard to test. Identifying the testing techniques to test the code is the main key here.

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

*Testing isn't everything:* There are approaches other than testing to create better software. Methods other than testing include:

**Inspection Methods:** Methods like walkthroughs, desk checking, formal inspections and code reading appear to beas effective as testing but the bugs caught do not 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.

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

7. **The pesticide paradox and the complexity barrier**
   Whatever methods we are using to remove completely bugs are notremoving but methods become ineffectual due to subtler bugs Complexity should also be manageable.

**DICHOTOMIES:**

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

This table shows few important differences between testing anddebugging.

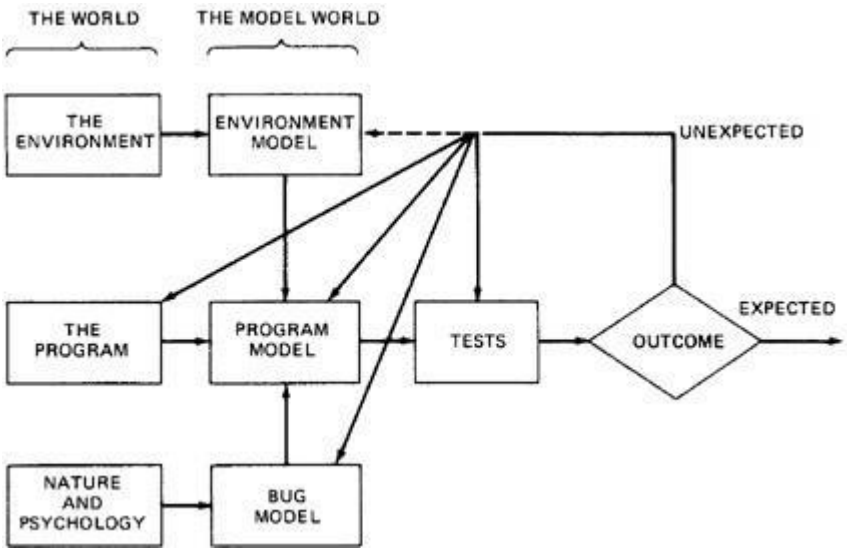
| **Testing** | **Debugging** |
|---|---|
| Testing starts with known conditions, uses predefined procedures and has predictable outcomes. | Debugging starts from possibly unknown initial conditions and the end cannot be predicted except statistically. |
| Testing can and should be planned, designed and | Procedure and duration of debugging |

| | |
|---|---|
| scheduled. | cannot be so constrained. |
| Testing is a demonstration oferror or apparent correctness. | Debugging is a deductive process. |
| Testing proves a programmer'sfailure. | Debugging is the programmer's vindication (Justification). |
| Testing, as executes, should strive to be predictable, dull, constrained, rigid and inhuman. | Debugging demands intuitive leaps, experimentation and freedom. |
| Much testing can be done without design knowledge. | Debugging is impossible without detailed design knowledge. |
| Testing can often be done by an outsider. | Debugging must be done by an insider. |
| Much of test execution and design can be automated. | Automated debugging is still a dream. |

- **Function versus Structure:** Tests can be designed from a functional or a structural point of view. In **functional testing**, the program or system is treated as a black-box. It is subjected to inputs, and its outputs are verified for conformance to specified behavior. Functional testing takes the user point of view- bother about functionality and features and not the program's implementation. **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 mergeinto 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 organizedinto 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 inthe 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 andthe users' complaints?

**MODEL FOR TESTING:**




It includes three models: A model of the environment, a model of the programand 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 withand 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 smartto 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 testit.

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 ofwhat 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 Within a component affects only that component's behavior.
3. **Control Bug Dominance:** The belief those errors in the control structures (if, switch etc) of programsdominate 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 is at code design.

**TESTS**

Tests are formal procedures, Inputs must be prepared, Outcomes should predict, tests should be documented, commands need to be executed, and results are to be observed. *All these errors are subjected to error*

**We do three distinct kinds of testing on a typical softwaresystem. They are:**

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

**Integration Testing: Integration** is the process by which components are aggregated to create larger components. **Integration Testing** is testing done to show that eventhough the components were individually satisfactory (afterpassing component testing), checks the combination of components are incorrect or inconsistent.

**System Testing:** A **System** is a big component. **System Testing** is aimed at revealing bugs that cannot be attributed to components. It includes testing for performance, security, accountability, configuration sensitivity, startup and recovery.

***CONSEQUENCES OF BUGS:***

**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 moreattention 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 ofinstallations: 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= ($) = Frequency * (Correction cost + Installation cost + Consequential cost)***

**CONSEQUENCES OF BUGS:** The consequences of a bug can be measure in terms of human rather than machine. Some consequences of a bug on a scale of one to ten are:

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 behavior because of the bug is dehumanizing. *E.g.* Names are truncated arbitrarily modified.
4. **Disturbing:** It refuses to handle legitimate (authorized / legal) transactions. The ATM won't 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 outof our hands because the system fails.
10. **Infectious:** What can be worse than a failed system? One that corrupt other systems even though it does not fall in itself; that erodes the social

Physical environment; that melts nuclear reactors and startswar.

**FLEXIBLE SEVERITY RATHER THAN ABSOLUTES:**

Quality can be measured as a combination of factors, ofwhich number of bugs and their severity is only one component.

Many organizations have designed and used satisfactory,quantitative, quality metrics.

Because bugs and their symptoms play a significant role in such metrics, as testing progresses, you see the quality rise to a reasonable value which is deemed to be safe to ship the product.

The factors involved in bug severity are:

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

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

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

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

**The software development phase:** Severity depends on development phase. Any bugs gets more severe as

it gets closer to field use and more severe the longerit has been around.

***TAXONOMY OF BUGS:***

- There is no universally correct way categorize bugs. The taxonomy isnot rigid.
- A given bug can be put into one or another category depending on itshistory 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 DesignBugs.

**REQUIREMENTS, FEATURES AND FUNCTIONALITY BUGS:**

Various categories in Requirements, Features and Functionality 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 percentages to more than 50%, depending on the application andenvironment.
   - What hurts most about the bugs is that they are the earliest to invade the system and thelast 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, andfoster more bugs.

3. **Feature Interaction Bugs:**
   - Providing correct, clear, implementable and testable feature specifications is not enough.
   - Features usually come in groups or related features. The features of each group and the interaction of features with in the group are usually well tested.
   - The problem is unpredictable interactions between feature groups or even between individual features. For example, your telephone is provided with call holding and call forwarding. The interactions between these two features may have bugs.
   - Every application has its peculiar set of features and a much bigger set of unspecified feature interaction potentials and therefore result in feature interaction bugs.

**Specification and Feature Bug Remedies:**

- Most feature bugs are rooted in human to human communication problems. One solution is to use high-level, formal specification languages or systems.
- Such languages and systems provide short term support but in the long run, do not solve the problem.
- ***Short term Support:*** Specification languages facilitate formalization of requirements and inconsistency and ambiguity analysis.
- ***Long term Support:*** Assume that we have a great specification language and that can be used to create unambiguous, complete specifications with unambiguous complete tests and consistent test criteria.

The specification problem has been shifted to a higher level

- But not eliminated.

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

**STRUCTURAL BUGS:** Various categories in Structuralbugs include:

**Control and Sequence Bugs:**

Control and sequence bugs include paths left out, unreachable code, improper nesting of loops, loop-back or loop termination criteria incorrect, missing process steps, duplicated processing, unnecessary processing, rampaging, GOTO's, ill-conceived (not properly planned) switches, spaghetti code, and worst of all, pachinko code.

One reason for control flow bugs is that this area is amenable (supportive) too theoretical treatment.

Most of the control flow bugs are easily tested and caught in unit testing.

Another reason for control flow bugs is that use of old code especially ALP & COBOL code are dominated by control flow bugs.

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

**Logic Bugs:**

Bugs in logic, especially those related to misunderstanding how case statements and logic operators behave singly and combinations

Also includes evaluation of Boolean expressions in deeply nested IF- THEN-ELSE constructs.

If the bugs are parts of logical (i.e. Boolean) processing not related to control flow, they are characterized as processing bugs.

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

**Processing Bugs:**

Processing bugs include arithmetic bugs, algebraic, mathematical function evaluation, algorithm selection and general processing.

Examples of Processing bugs include: Incorrect conversion from one data representation to other, ignoring overflow, improper use of greater-than-or-equal etc

Although these bugs are frequent (12%), they tend to be caught in good unit testing.

**Initialization Bugs:**

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

Superfluous bugs are generally less harmful but can affectperformance.

Typical initialization bugs include: Forgetting to initialize the variables before first use, assuming that they are initialized elsewhere, initializing to the wrong format, representation or type etc

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

**Data-Flow Bugs and Anomalies:**

Most initialization bugs are special case of data flow anomalies.

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

**DATA BUGS:**

Data bugs include all bugs that arise from the specification of data objects, their formats, the number of such objects, and their initial values.

Data Bugs are at least as common as bugs in code, but they are often treated as if they did not exist at all.

*Code migrates data:* Software is evolving towards programs in which more and more of the control and processing functions are stored in tables.

Because of this, there is an increasing awareness that bugs in code are only half the battle and the data problems should be given equalattention.

**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) NoClean up

Static Data are fixed in form and content. They appear in the source code or database directly or indirectly, for example a number, astring of characters, or a bit pattern.

Compile time processing will solve the bugs caused by static data.

**Information, parameter, and control:** Static or dynamic

1. Data can serve in one of three roles, or in combination of roles: as a parameter, for control, or for information.

**Content, Structure and Attributes: Content** can be an actual bit pattern, character string, or number put into a data structure. Content is a pure bit pattern and has no meaning unless it is interpreted by a hardware or software processor. All data bugs result in the corruption or misinterpretation of content. **Structure** relates to the size, shape and numbers that describe the data object that is memory location used to store the content. (E.g. A two dimensional array). **Attributes** relates to the specification meaning that is the semantics associated with the contents of a data object.

**CODING BUGS:**

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

- Syntax errors are generally not important in the scheme of things if the source language translator has adequate syntax checking.
- If a program has many syntax errors, then we shouldexpect many logic and coding bugs.
- The documentation bugs are also considered as coding bugs which may mislead the maintenanceprogrammers.
- 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 is communicating routines.

The external environment is fixed and the system must adapt to it but the internal environment, which consists of interfaces with other components, can be negotiated.

Internal interfaces have the same problem as external interfaces.

**Hardware Architecture:**

Bugs related to hardware architecture originate mostly from misunderstanding how the hardware works.

Examples of hardware architecture bugs: address generation error, i/o device operation / instruction error, waiting too long fora 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 amisunderstanding 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 localizethem 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 astress test are effective methods for these bugs.

**Control and Sequence Bugs (Systems Level):**

These bugs include: Ignored timing, Assuming that events occurin 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 intomemory 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 preventsbugs is always preferable to a test method that discovers them.

The design remedy in resource management is to keep the resource structure simple: the fewest different kinds of resources, the fewest pools, and no private resource management.

**Integration Bugs:**

Integration bugs are bugs having to do with the integration of, and with the interfaces between, working and tested components.

These bugs results from inconsistencies or incompatibilities between components.

The communication methods include data structures, call sequences, registers, semaphores, and communication links and protocols results in integration bugs.

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

**System Bugs:**

System bugs covering all kinds of bugs that cannot be ascribed to a component or to their simple interactions, but result from the totality of interactions between many components such as programs, data, hardware, and the operating systems.

There can be no meaningful system testing until there has beenthorough component and integration testing.

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

**TEST AND TEST DESIGN BUGS:**

Testing: testers have no immunity to bugs. Tests require complicatedscenarios and databases.

They require code or the equivalent to execute and consequently theycan 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 criterion has to be designed. The more complicated the criteria, the likelier they are to have bugs.

**Remedies:** The remedies of test bugs are:

**Test Debugging:** The first remedy for test bugs is testing and debugging the tests. Test debugging, when compared to program debugging, is easier because tests, when properly designed are simpler than programs

And do not have to make concessions to efficiency.

**Test Quality Assurance:** Programmers have the right to ask howquality 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 testexecution 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.

**Unit – II**

***Objective of the Unit***

- Understand the concept of path testing.
- Identify the components of a control flow diagram and compare thesame with a flowchart.
- Interpret a control flow-graph and demonstrate the complete pathtesting to achieve C1+C2.
- Classify the predicates and variables as dependant/independent and correlated/uncorrelated.
- Understand the path sensitizing and Instrumentation methods andclassify whether the path is achievable or not.

***Topic:*** **FLOWGRAPHS AND PATH TESTING:**

- ***Motivation (Why this topic is significant for the discussion?):***
- if it is difficult to prove whether the program has bugs or not can be easy to prove in terms of flow graph and flowchart by applying Path sensitization and path Instrumentation and all path concepts.

- ***Notes:*** **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 isindispensable.

- **CONTROL FLOW GRAPHS:**
  - The control flow graph is a graphical representation of aprogram's control structure. It uses the elements named process blocks, decisions, and junctions.
  - The flow graph is similar to the earlier flowchart, with which itis 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 isexecuted, 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 consist of a single statement or instruction, a sequence of statements or instructions, a single entry/exit subroutine, a macro or function call, or a sequence of these.
    2. **Decisions:**
       - A decision is a program point at which the control flow can diverge.
       - Machine language conditional branch and conditional skip instructions are examples ofdecisions.
       - 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 andCase Statements
    4. **Junctions:**
       - A junction is a point in the program where the control flow can merge.
       - Examples of junctions are: the target of a jump or skip instruction in ALP, a label that is a target of GOTO.
- **CONTROL FLOW GRAPHS Vs FLOWCHARTS:**
  - A program's flow chart resembles a control flow graph.
  - In flow graphs, we don't show the details of what is in aprocess block.
  - In flow charts every part of the process block is drawn.
  - The flowchart focuses on process steps, where as the flowgraph focuses on control flow of the program.

- The act of drawing a control flow graph is a useful tool thatcan help us clarify the control flow and data flow issues.

To understand this, we will go through an example written in a FORTRAN like programming language called **Programming Design Language (PDL)**

**FLOWGRAPH AND FLOWCHART GENERATION:**

Flowcharts can be

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

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

0. **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.
1. A path may go through several junctions, processes, or decisions, one or more times.

Paths consist of segments.

The segment is a link - a single process that lies between two nodes.

0. A path segment is succession of consecutive links that belongs to some path.
1. The length of path measured by the number of links in it and not by the number of the instructions or statements executed along that path.

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

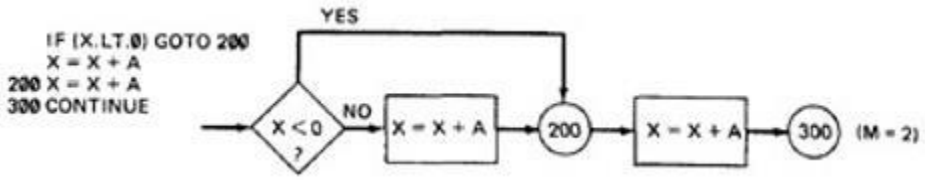
**FUNDAMENTAL PATH SELECTION CRITERIA:**

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

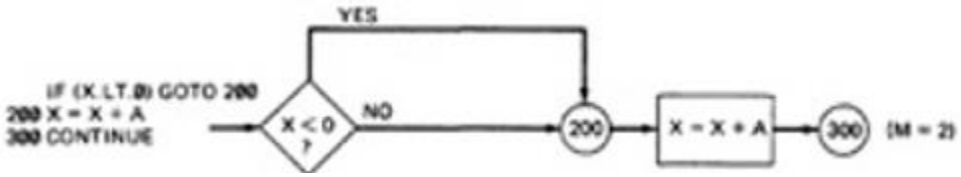
Defining complete testing:

0. Exercise every path from entry to exit
1. Exercise every statement or instruction at least once
2. Exercise every branch and case statement, in each direction at least once
3. If prescription 1 is followed then 2 and 3 areautomatically followed. But it is impractical for most routines. It can be done for the routines that have no loops, in which it is equivalent to 2 and 3 prescriptions.

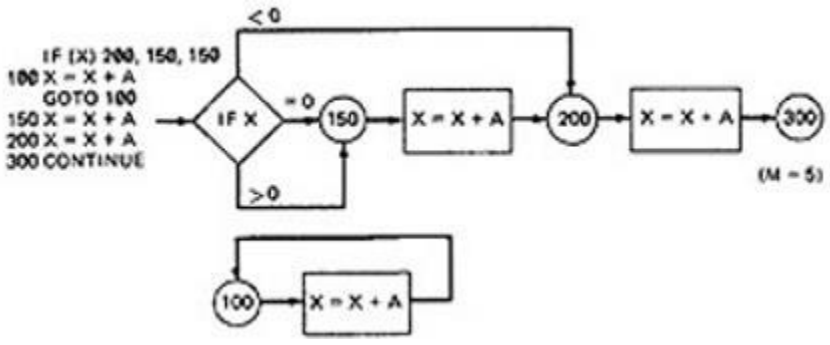
**EXAMPLE: Here is the correct version.**




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




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



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

**PATH TESTING CRITERIA:**

0. Any testing strategy based on paths must at least both exercise every instruction and take branches in all directions.
1. A set of tests that does this is not complete in anabsolute sense, but it is complete in the sense that anything less must leave something untested.
2. So we have explored three different testing criteria or strategies out of a potentially infinite family of strategies.
3. **Path Testing (Pinf)**:

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

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

**Statement Testing (P1)**:

Execute all statements in the program at least once under some test.If we do enough tests to achieve this, we are said to have achieved 100% statement coverage.

An alternate equivalent characterization is to say that we have achieved 100% node coverage. We denote this by C1.

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

**Branch Testing (P2)**:

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

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

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

For structured software, branch testing and therefore branch coverage strictly includes statement coverage.

We denote branch coverage by C2.

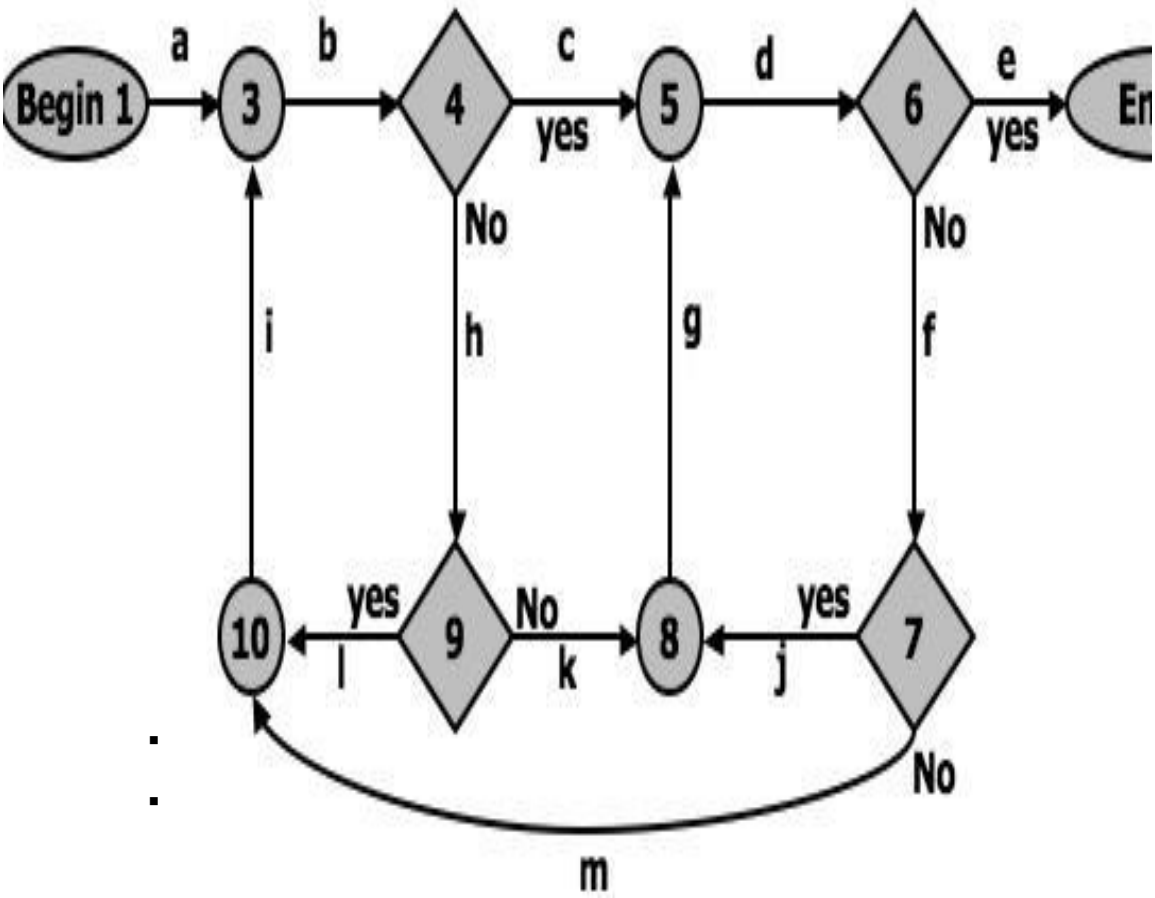
**Commonsense and Strategies:**

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

The question "why not use a judicious sampling of paths?, what is wrong with leaving some code, untested?" is ineffectual in the view of common sense and experience since: **(1.)** Not testing a piece of a code leaves a residue of bugs in the program in proportion to the size of the untested code and the probability of bugs. **(2.)** The high probability paths are always thoroughly tested if only to demonstrate that the system works properly.

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

**APPLICATION OF PATH TESTING:**



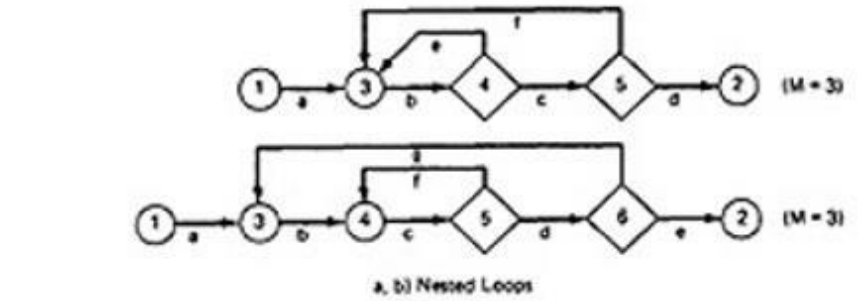
- 
- 
- 

**Practical Suggestions in Path Testing:**

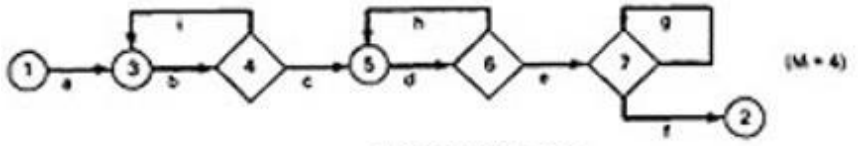
0. Draw the control flow graph on a single sheet of paper.
1. Make several copies - as many as you will need for coverage (C1+C2) and several more.
2. Use a yellow highlighting marker to tracepaths. Copy the paths onto a master sheets.
3. Continue tracing paths until all lines on the master sheet are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

| 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 | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | ✓ | | | | ✓ |

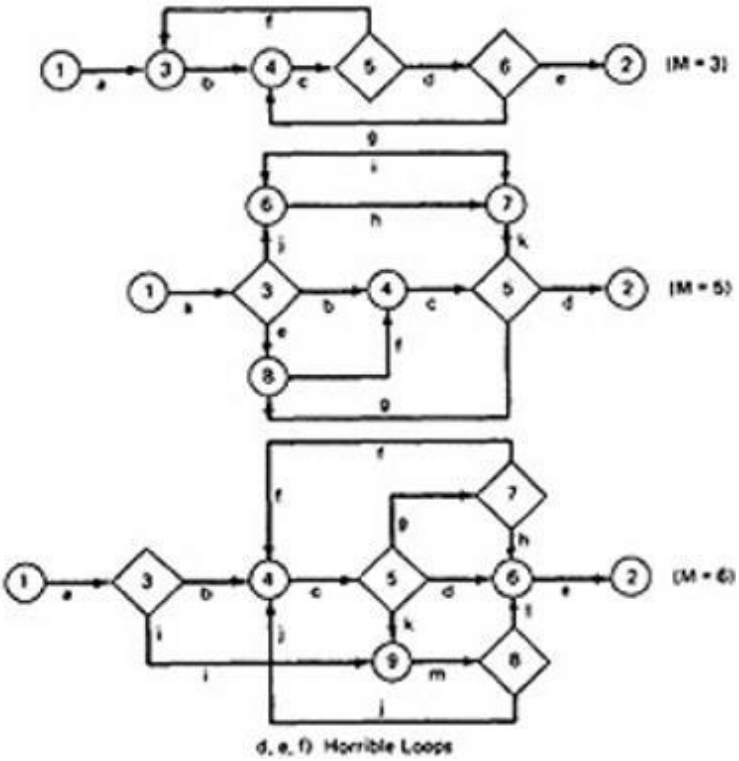
**LOOPS:**




a, b) Nested Loops




c) Concatenated Loops




d, e, f) Horrible Loops

### 1. 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, and 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 .

**PREDICATES, PATH PREDICATES AND ACHIEVABLE PATHS:**

**PREDICATE:** The logical function evaluated at a decision is called Predicate. The direction taken at a decision depends onthe 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 theroutine to take a specific path.

**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 notdepend on processing.

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

Every path through a routine is achievable only if all the predicates in that routine are uncorrelated.

Interpretation means in the predicates all the variable values need to besubstituted in order to get interms of solely input vectors.

**PATH PREDICATE EXPRESSIONS:**

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

**TESTING BLINDNESS:**

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

**Assignment Blindness:**

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

**Equality Blindness:**

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

**Self-Blindness:**

Self blindness occurs when the buggy predicate is a multiple ofthe correct predicate and as a result is indistinguishable alongthat path.

***PATH SENSITIZING:***

**REVIEW: ACHIEVABLE AND UNACHIEVABLE PATHS:**

We want to select and test enough paths to achieve a satisfactorynotion of test completeness such as C1+C2.

Extract the programs control flowgraph and select a set of tentativecovering paths.

For any path in that set, interpret the predicates along the path asneeded to express them in terms of the input vector. In general

individual predicates are compound or may become compound as aresult 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).***

***ADFGHIJKL+AEFGHIJKL+BCDFGHIJKL+BCEFGHIJKL***

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

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

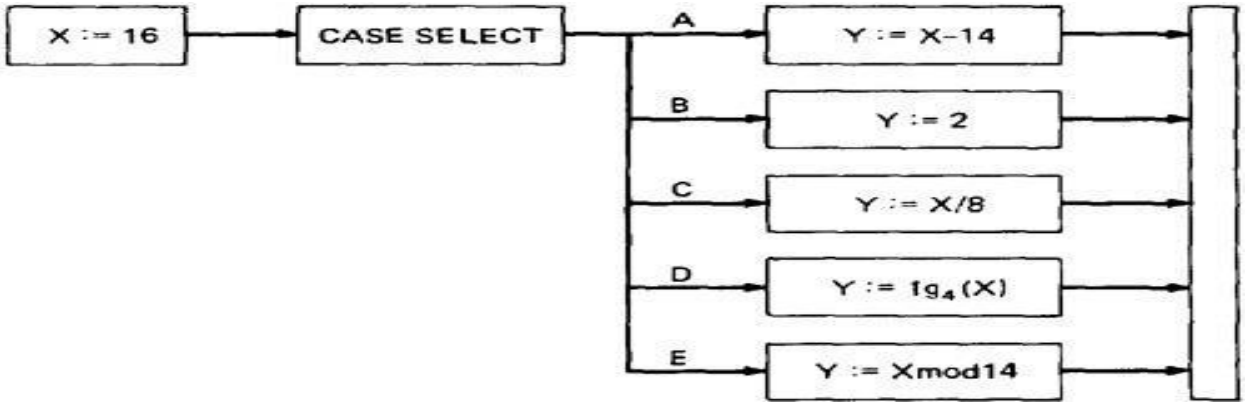
If you can find a solution, then the path is achievable.

If you cant find a solution to any of the sets of inequalities, the path isun achievable.

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

**PATH INSTRUMENTATION:** This is what we have to do to confirm thatdesired outcome for the intended path.

**Co-incidental Correctness:** The coincidental correctness stands for achieving the desired outcome for wrong reason




For all the cases the outcome is two ,this is not aim of designing casestatements, each case should give different outcome that is the intention

of designing case statement but here fails to avoid this we will move toPath Instrumentation.

The types of instrumentation methods include:

**Interpretive Trace Program:**

Tracing the path and calculating the intermediate value so that canachieve path instrumentation.

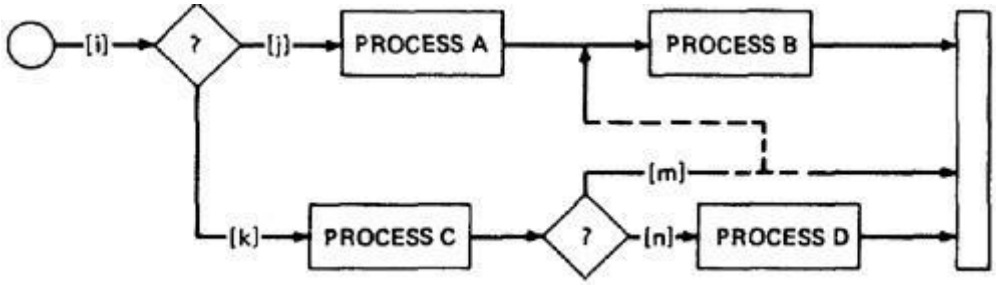
**Traversal Marker or Link Marker:**

A simple and effective form of instrumentation is called a traversalmarker or link marker.

Name every link by a lower case letter.

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

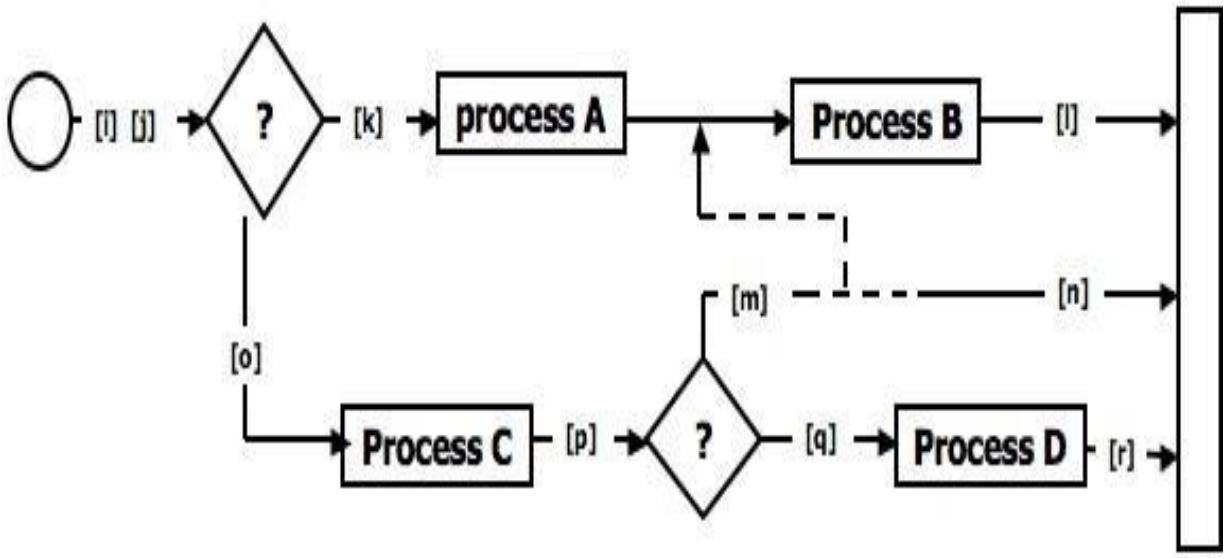
Why single link marker is not enough




Due to Ramping goto instead of going ikm path it goes to ikm through process B also give same result. So it fails to avoid this double link markers are implemented.

**Two Link Marker Method:**

Adding the link markers at the begin and at the end of the path also.Then correctly identify which path it is.




**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*:***

Integration, Coverage, and paths in called Components: It must well worked forUnit level path testing

**New code:** wholly new or substantially modified code should always besubjected enough path testing to achieve C2.

**Rehosting**: Software is rehosted because it is no longer cost-effective tosupport the environment in which it runs

The objective of software rehosting is to change the environment and not therehosted software.

**Maintenance**: maintenance test methods into efficient methodologies thatprovide the kind of coverage should achieve in maintenance.

**Unit-III**

***Objective of the unit:***

- students understand what are the states of the variables, what are the reasons to occur anomalies , they can realize and write programs withoutoccurring of bugs.

***Topic: Data-Flow Testing***

***Motivation:***

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

- ***Notes:*** **DATA FLOW MACHINES:**
  - There are two types of data flow machines with differentarchitectures. (1) Von Neumann machines (2) Multi- instruction, multi-data machines (MIMD).
  - **Von Neumann Machine Architecture:**
    - Most computers today are von-Neumann machines.
    - This architecture features interchangeable storage of instructions and data in the same memory units.
    - The Von Neumann machine Architecture executesone instruction at a time in the following, micro instruction sequence:
      1. Fetch instruction from memory
      2. Interpret instruction
      3. Fetch operands
      4. Process or Execute
      5. Store result
      6. Increment program counter
      7. GOTO 1
  - **Multi-instruction, Multi-data machines (MIMD) Architecture:**
    - These machines can fetch several instructions andobjects in parallel.
    - They can also do arithmetic and logical operations simultaneously on different data objects.
    - The decision of how to sequence them depends onthe compiler.
- **BUG ASSUMPTION:**
  - The bug assumption for data-flow testing strategies is that control flow is generally correct and that something has gone wrong with the software so that data objects are not available when they should be, or silly things are being done to data objects.

- Also, if there is a control-flow problem, we expect it to have symptoms that can be detected by data-flow analysis.
- Although we'll be doing data-flow testing, we won't be using data flow graphs as such. Rather, we'll use an ordinary control flow graph annotated to show what happens to the data objects of interest at the moment.

- **DATA FLOW GRAPHS:**
  - The data flow graph is a graph consisting of nodes and directed links.



  - We will use a control graph to show what happens to dataobjects of interest at that moment.
  - Our objective is to expose deviations between the data flowswe have and the data flows we want.
  - **Data Object State and Usage:**
    - Data Objects can be created, killed and used.
    - They can be used in two distinct ways: (1) In aCalculation (2) As a part of a Control Flow Predicate.
    - The following symbols denote these possibilities:
      1. **Defined:** d - defined, created, initialized etc
      2. **Killed or undefined:** k - killed, undefined,released etc
      3. **Usage:** u - used for something (c - used in Calculations, p - used in a predicate)
    - **1. Defined (d):**
      - An object is defined explicitly when it appears in a data declaration.
      - Or implicitly when it appears on the lefthand side of the assignment.
      - It is also to be used to mean that a file hasbeen opened.
      - A dynamically allocated object has beenallocated.
      - Something is pushed on to the stack.
      - A record written.

**2. Killed or Undefined (k):**

- An object is killed on undefined when it is released or otherwise made unavailable.
- When its contents are no longer known with certitude (with absolute certainty / perfectness).
- Release of dynamically allocated objects back to the availability pool.
- Return of records.
- The old top of the stack after it is popped.

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

**3. Usage (u):**

- A variable is used for computation (c) when it appears on the right hand side of an assignment statement.
- A file record is read or written.
- It is used in a Predicate (p) when it appears directly in a predicate.

**DATA FLOW ANOMALIES:**

An anomaly is denoted by a two-character sequence of actions.

For example, ku means that the object is killed and then used, where asdd means that the object is defined twice without an intervening usage.

What is an anomaly is depend on the application.

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

0. **dd** :- probably harmless but suspicious. Why definethe object twice without an intervening usage?
1. **dk** :- probably a bug. Why define the object withoutusing it?
2. **du** :- the normal case. The object is defined and thenused.
3. **kd** :- normal situation. An object is killed and thenredefined.
4. **kk** :- harmless but probably buggy. Did you want tobe sure it was really killed?
5. **ku** :- a bug. the object does not exist.
6. **ud** :- usually not a bug because the language permits reassignment at almost any time.
7. **uk** :- normal situation.
8. **uu** :- normal situation.

In addition to the two letter situations, there are six single lettersituations.

We will use a leading dash to mean that nothing of interest (d,k,u) occursprior to the action noted along the entry-exit path of interest.

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

They possible anomalies are:

0. **-k** :- possibly anomalous because from the entrance to this point on the path, the variable had not been defined. We are killing a variable that does not exist.
1. **-d** :- okay. This is just the first definition along this path.

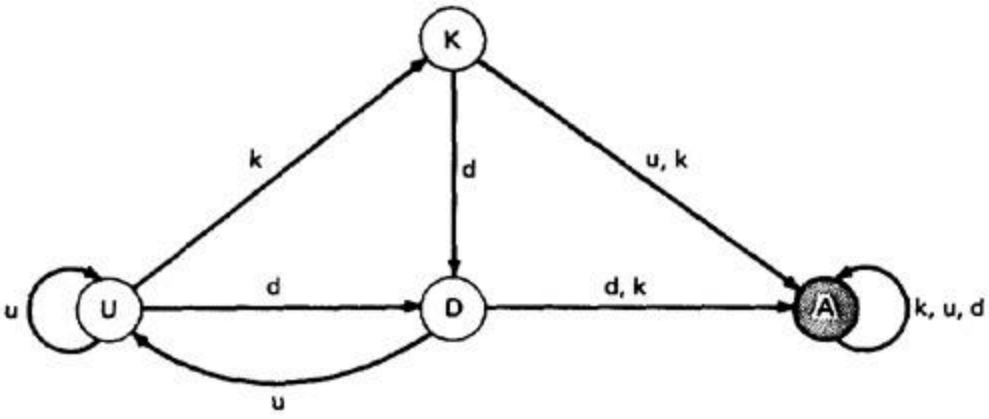
2. **-u** :- possibly anomalous. Not anomalous if the variable is global and has been previously defined.
3. **k-** :- not anomalous. The last thing done on this pathwas to kill the variable.
4. **d-** :- possibly anomalous. The variable was defined and not used on this path. But this could be a global definition.
5. **u-** :- not anomalous. The variable was used but not killed on this path. Although this sequence is not anomalous, it signals a frequent kind of bug. If d andk mean dynamic storage allocation and return respectively, this could be an instance in which a dynamically allocated object was not returned to the pool after use.

**DATA FLOW ANOMALY STATE GRAPH:**

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

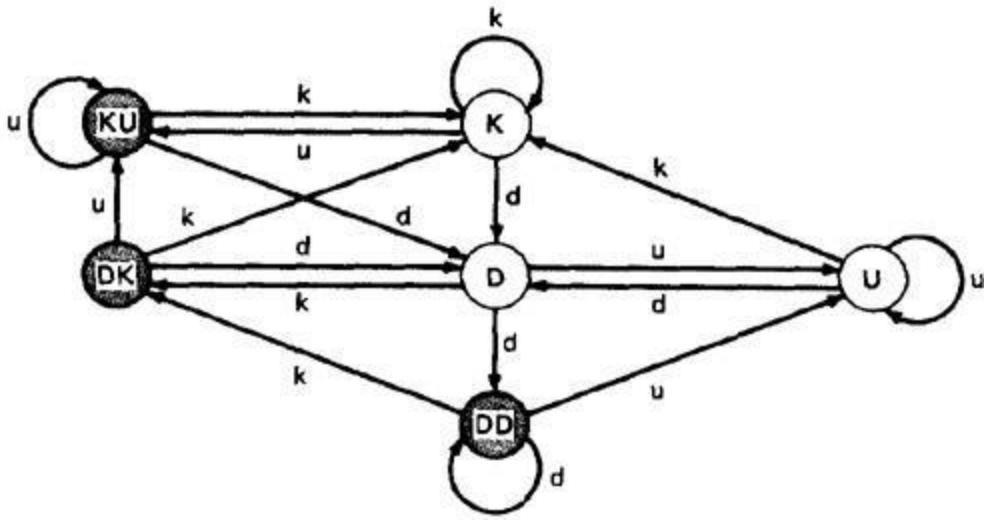
0. **K** :- undefined, previously killed, does not exist
1. **D** :- defined but not yet used for anything
2. **U** :- has been used for computation or in predicate
3. **A** :- anomalous

These capital letters (K,D,U,A) denote the state of the variable and shouldnot be confused with the program action, denoted by lower case letters.




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

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




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

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

- **STATIC Vs DYNAMIC ANOMALY DETECTION:**
  - Static analysis is analysis done on source code without actually executing it. For example: source code syntax error detection is the static analysis result.
  - Dynamic analysis is done on the fly as the program is being executed and is based on intermediate values that result from the program's execution. For example: a division by zero warning is the dynamic result.
  - **Why Static Analysis isn't enough?** There are many things for which current notions of static analysis are inadequate. They are:

- **Dead Variables:** Although it is often possible to prove that a variable is dead or alive at a given point in the program, the general problem is unsolvable.
- **Arrays:** Arrays are problematic in that the array is defined or killed as a single object, but reference is to specific locations within the array. Array pointers are usually dynamically calculated, so there's no way to do a static analysis to validate the pointer value. In many languages, dynamically allocated arrays contain garbage unless explicitly initialized and therefore, -u anomalies are possible.
- **Records and Pointers:** The array problem and the difficulty with pointers is a special case of multipart data structures. We have the same problem with records and the pointers to them. Also, in many applications we create files and their names dynamically and there's no way to determine,without execution, whether such objects are in the proper state on a given path or, for that matter,whether they exist at all.
- **Dynamic Subroutine and Function Names in a Call:** subroutine or function name is a dynamic variable in a call. What is passed, or a combinationof subroutine names and data objects, is constructedon a specific path. There's no way, without executing the path, to determine whether the call is correct or not.
- **False Anomalies:** Anomalies are specific to paths. Even a "clear bug" such as ku may not be a bug ifthe path along which the anomaly exists is unachievable. Such "anomalies" are false anomalies. Unfortunately, the problem of determining whether a path is or is not achievable is unsolvable.
- **Recoverable Anomalies and Alternate State Graphs:** What constitutes an anomaly depends on context, application, and semantics. How does the compiler know which model I have in mind? It can't because the definition of "anomaly" is not fundamental. The language processor must have a built-in anomaly definition with which you may or may not (with good reason) agree.
- **Concurrency, Interrupts, System Issues:** As soon as we get away from the simple single-task uniprocessor environment and start thinking in terms of systems, most anomaly issues become vastly more complicated. How often do we define or

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

routine.

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

- **DATA FLOW MODEL:**
  - The data flow model is based on the program's control flow graph - Don't confuse that with the program's data flow graph.
  - Here we annotate each link with symbols (for example, d, k, u, c, p) or sequences of symbols (for example, dd, du, ddd)that denote the sequence of data operations on that link with respect to the variable of interest. Such annotations are calledlink weights.
  - The control flow graph structure is same for every variable: it is the weights that change.
  - **Components of the model:**
    1. To every statement there is a node, whose name is unique. Every node has at least one out link and at least one in link except for exit nodes and entrynodes.
    2. Exit nodes are dummy nodes placed at the outgoing arrowheads of exit statements (e.g., END, RETURN), to complete the graph. Similarly, entry nodes are dummy nodes placed at entry statements (e.g., BEGIN) for the same reason.

The out link of simple statements (statements with only one out link)is weighted by the proper sequence of data-flow actions for that statement. Note that the sequence can consist of more than one letter

3. Predicate nodes (e.g., IF-THEN-ELSE, DO WHILE, CASE) are weighted with the p - use(s) on everyoutlink, appropriate to that outlink.
4. Every sequence of simple statements (e.g., a sequence of nodes with one inlink and one outlink)

   can be replaced by a pair of nodes that has, asweights on the link between them, the concatenation of link weights.

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

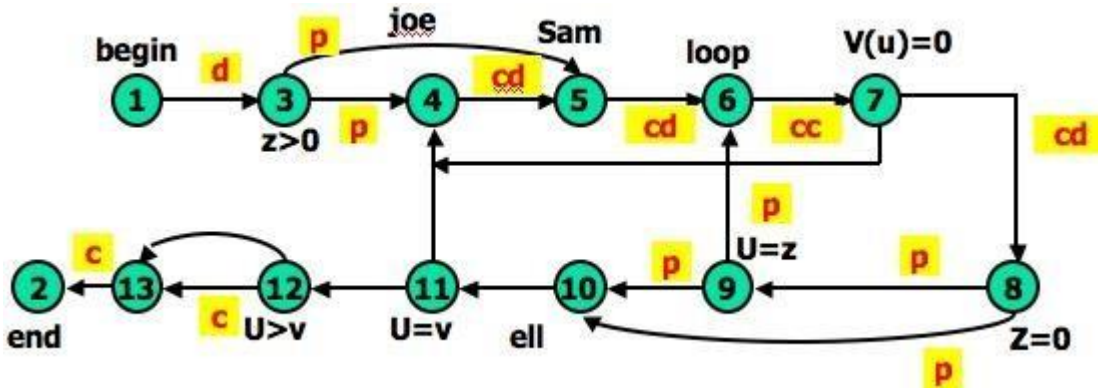
6. Conversely, a link with several data-flow actions on it can be replaced by a succession of equivalent links, each of which has at most one data-flow action for any variable.

Let us consider the example:

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




**STRATEGIES OF DATA FLOW TESTING:**

- **TERMINOLOGY:**

  **Definition-Clear Path Segment**, with respect to variable X, isa connected sequence of links such that X is (possibly) defined on the first link and not redefined or killed on any subsequent link of that path segment.

  **Loop-Free Path Segment** is a path segment for which every node in it is visited atmost once. For Example, path (4,5,6,7,8,10)

1. **Simple path segment** is a path segment in which at most one node is visited twice. For example, in Figure 3.10, (7,4,5,6,7) is a simple path segment. A simple path segment iseither loop-free or if there is a loop, only one node is involved.
2. A **du path** from node i to k is a path segment such that if the last link has a

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

**STRATEGIES:** The structural test strategies discussed below are based on the program's control flow graph. They differ in the extent to which predicateuses and/or computational uses of variables are included in the test set. Various types of data flow testing strategies in decreasing order of their effectiveness are:

**All - du Paths (ADUP):** The all-du-paths (ADUP) strategy is the strongest data-flow testing strategy discussed here. It

requires that every du path from every definition of every variable to every use of that definition be exercised under some test.

0. The all-du-paths strategy is a strong criterion, but it does not take as many tests as it might seem at first because any one test simultaneously satisfies the criterion for several definitions and uses of several different variables.

**All Uses Strategy (AU):**The all uses strategy is that at least one definition clear path from every definition of every variable to every use of that definition be exercised under some test. Just as we reduced our ambitions by stepping down from all paths (P) to branch coverage (C2), say, we can reduce the number of test cases by asking that the test set should include at least one path segment from every definition to every use that can be reached by that definition.

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

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

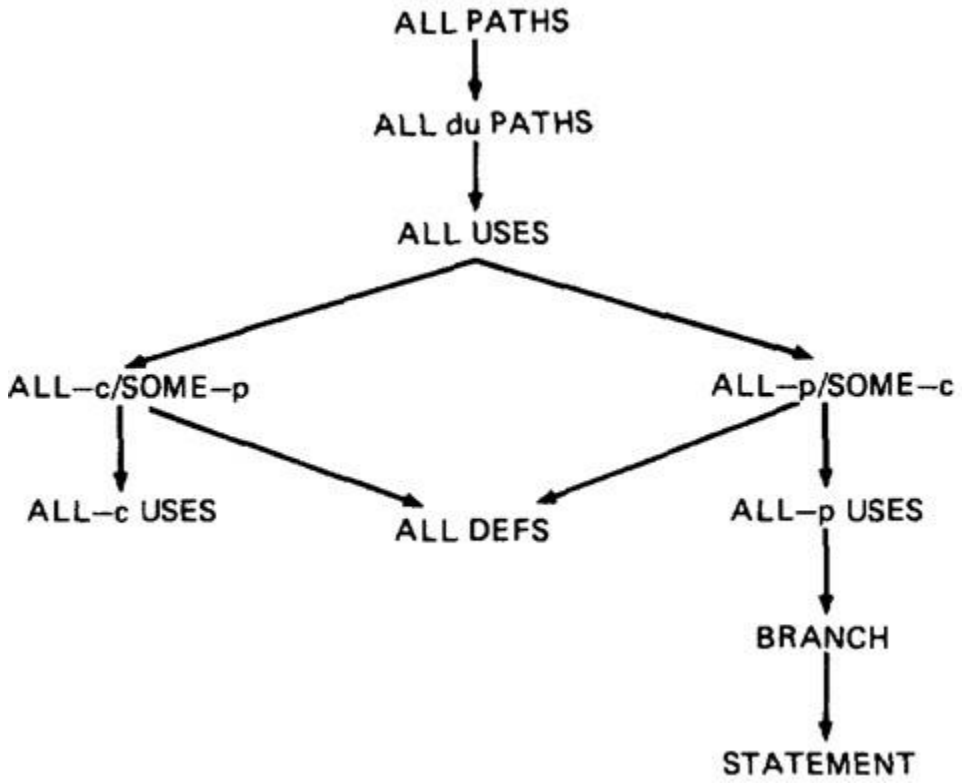
**All Definitions Strategy (AD) :** The all definitions strategy asks onlyevery definition of every variable be covered by at least one use of that variable, be that use a computational use or a predicate use.

1. **All Predicate use (APU), All Computational Uses (ACU) Strategies :** The all predicate uses strategy is derived from APU+C strategy by dropping the requirement that we include a c-use for the variable if there

are no p-uses for the variable. The all computational uses strategy is derived from ACU+P strategy by dropping the requirement that we include a p-use for the variable if there are no c-uses for the variable.

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

**ORDERING THE STRATEGIES:**




- The right-hand side of this graph, along the path from "all paths" to "all statements" is the more interesting hierarchy forpractical applications.
- Note that although ACU+P is stronger than ACU, both are incomparable to the predicate-biased strategies. Note also that "all definitions" is not comparable to ACU or APU.

**SLICING AND DICING:**

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

- A program **dice** is a part of a slice in which all statements which are known to be correct have been removed.
- In other words, a dice is obtained from a slice by incorporating information obtained through testing or experiment (e.g., debugging).

**Unit IV**

**Logic Based Testing**

**INTRODUCTION:**

- The functional requirements of many programs can be specified by **decision tables**, which provide a useful basis for program and test design.
- Consistency and completeness can be analyzed by using boolean algebra, which can also be used as a basis for test design. Boolean algebra is trivialized by using **Karnaugh-Veitch charts**.
- "Logic" is one of the most often used words in programmers' vocabularies but one of their least used techniques.
- Boolean algebra is to logic as arithmetic is to mathematics. Without it, the tester or programmer is cut off from many test and design techniques and tools that incorporate those techniques.
- Logic has been, for several decades, the primary tool of hardware logic designers.
- Many test methods developed for hardware logic can be adapted to software logic testing. Because hardware testing automation is 10 to 15 years ahead of software testing automation, hardware testing methods and its associated theory is a fertile ground for software testing methods.
- As programming and test techniques have improved, the bugs have shifted closer to the process front end, to requirements and their specifications. These bugs range from 8% to 30% of the total and because they're first-in and last-out, they're the costliest of all.
- The trouble with specifications is that they're hard to express.
- Boolean algebra (also known as the sentential calculus) is the most basic of all logic systems.
- Higher-order logic systems are needed and used for formal specifications.
- Much of logical analysis can be and is embedded in tools. But these tools incorporate methods to simplify, transform, and check specifications, and the methods are to a large extent based on boolean algebra.
- **KNOWLEDGE BASED SYSTEM:**
    - The **knowledge-based system** (also expert system, or "artificial intelligence" system) has become the programming construct of choice for many applications that were once considered very difficult.
    - Knowledge-based systems incorporate knowledge from a knowledge domain such as medicine, law, or civil engineering into a database. The data can then be queried and interacted with to provide solutions to problems in that domain.
    - One implementation of knowledge-based systems is to incorporate the expert's knowledge into a set of rules. The user can then provide data and ask questions based on that data.
    - The user's data is processed through the rule base to yield conclusions (tentative or definite) and requests for more data. The processing is done by a program called the **inference engine**.
    - Understanding knowledge-based systems and their validation problems requires an understanding of formal logic.
- Decision tables are extensively used in business data processing; Decision-table preprocessors as extensions to COBOL are in common use; boolean algebra is embedded in the implementation of these processors.
- Although programmed tools are nice to have, most of the benefits of boolean algebra can be reaped by wholly manual means if you have the right conceptual tool: the Karnaugh-Veitch diagram is that conceptual tool.

- **DECISION TABLES**
- Figure 6.1 is a limited - entry decision table. It consists of four areas called the condition stub, the condition entry, the action stub, and the action entry.
- Each column of the table is a rule that specifies the conditions under which the actions named in the action stub will take place.
- The condition stub is a list of names of conditions.

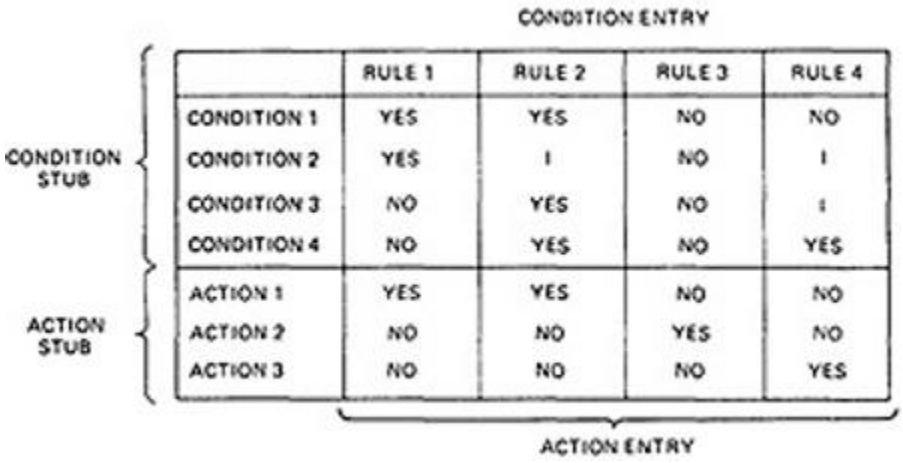



***Figure 6.1 : Examples of Decision Table.***

- A rule specifies whether a condition should or should not be met for the rule to be satisfied. "YES" means that the condition must be met, "NO" means that the condition must not be met, and "I" means that the condition plays no part in the rule, or it is immaterial to that rule.
- The action stub names the actions the routine will take or initiate if the rule is satisfied. If the action entry is "YES", the action will take place; if "NO", the action will not take place.
- 
- Figure 6.1 translated as
- Action 1 will take place if conditions 1 and 2 are met and if conditions 3 and 4 are not met (rule 1) or if conditions 1, 3, and 4 are met (rule 2).
- "Condition" is another word for predicate.
- Decision-table uses "condition" and "satisfied" or "met". Let us use "predicate" and TRUE / FALSE.
- Now the above translations become:
    1. Action 1 will be taken if predicates 1 and 2 are true and if predicates 3 and 4 are false (rule 1), or if predicates 1, 3, and 4 are true (rule 2).
    2. Action 2 will be taken if the predicates are all false, (rule 3).
    3. Action 3 will take place if predicate 1 is false and predicate 4 is true (rule 4).

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

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

***Figure 6.3 : The default rules of Table in Figure 6.1***

- **DECISION-TABLE PROCESSORS:**
  - Decision tables can be automatically translated into code and, as such, are a higher-order language
  - If the rule is satisfied, the corresponding action takes place
  - Otherwise, rule 2 is tried. This process continues until either a satisfied rule results in an action or no rule is satisfied and the default action is taken
  - Decision tables have become a useful tool in the programmers kit, in business data processing.
- **DECISION-TABLES AS BASIS FOR TEST CASE DESIGN:**
  1. The specification is given as a decision table or can be easily converted into one.
  2. The order in which the predicates are evaluated does not affect interpretation of the rules or the resulting action - i.e., an arbitrary permutation of the predicate order will not, or should not, affect which action takes place.
  3. The order in which the rules are evaluated does not affect the resulting action - i.e., an arbitrary permutation of rules will not, or should not, affect which action takes place.
  4. Once a rule is satisfied and an action selected, no other rule need be examined.
  5. If several actions can result from satisfying a rule, the order in which the actions are executed doesn't matter
- **DECISION-TABLES AND STRUCTURE:**
  - Decision tables can also be used to examine a program's structure.
  - Figure 6.4 shows a program segment that consists of a decision tree.
  - These decisions, in various combinations, can lead to actions 1, 2, or 3.

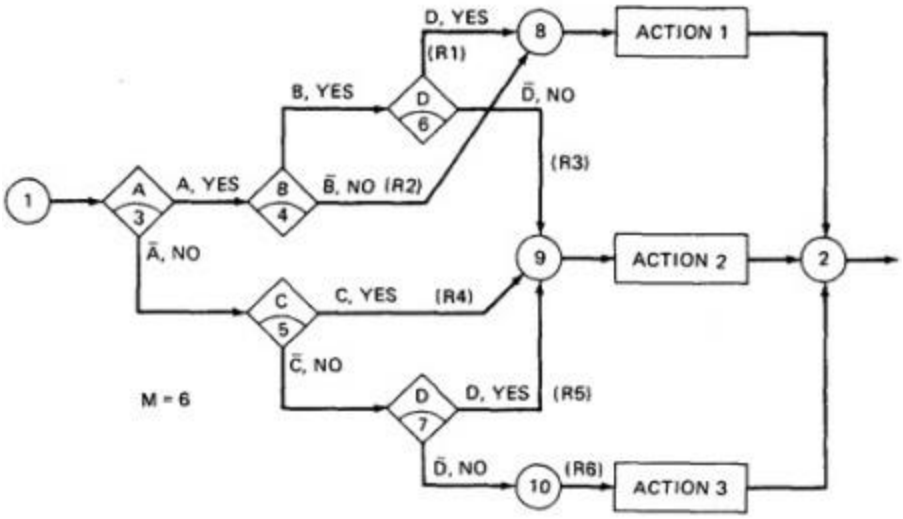



***Figure 6.4 : A Sample Program***

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

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

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

| RULE 1.1 | RULE 1.2 | RULE 2.1 | RULE 2.2 |
|---|---|---|---|
| YES | YES | YES | YES |
| YES | NO | NO | NO |
| YES | YES | YES | NO |
| NO | NO | NO | NO |
| YES | YES | NO | NO |
| NO | NO | YES | YES |

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

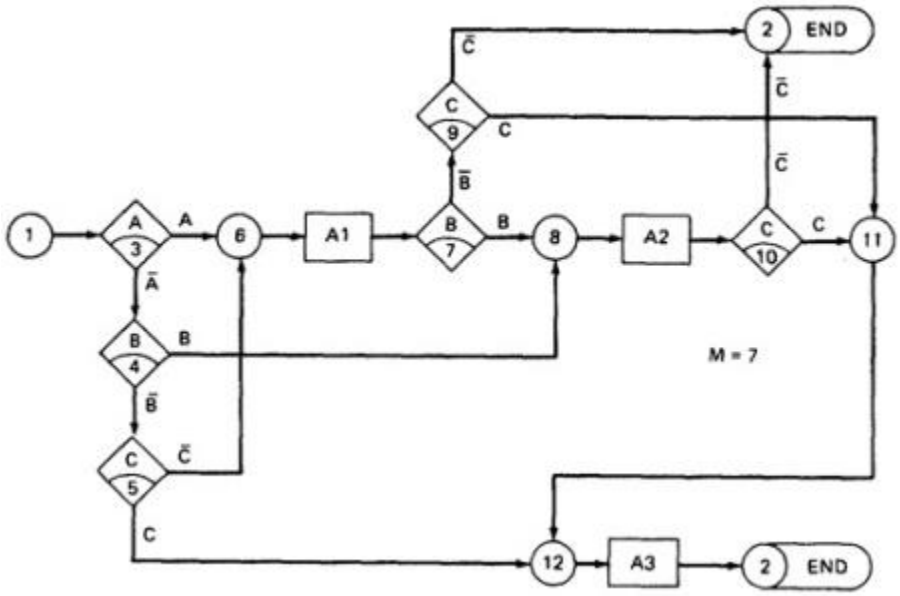
- Sixteen cases are represented in Table 6.1, and no case appears twice.
- Consequently, the flowgraph appears to be complete and consistent.
- As a first check, before you look for all sixteen combinations, count the number of Y's and N's in each row. They should be equal. We can find the bug that way.

- **ANOTHER EXAMPLE - A TROUBLE SOME PROGRAM:**

  - Consider the following specification whose putative flowgraph is shown in Figure 6.5:
    1. If condition A is met, do process A1 no matter what other actions are taken or what other conditions are met.
    2. If condition B is met, do process A2 no matter what other actions are taken or what other conditions are met.

3. If condition C is met, do process A3 no matter what other actions are taken or what other conditions are met.
4. If none of the conditions is met, then do processes A1, A2, and A3.
5. When more than one process is done, process A1 must be done first, then A2, and then A3. The only permissible cases are: (A1), (A2), (A3), (A1,A3), (A2,A3) and (A1,A2,A3).

- Figure 6.5 shows a sample program with a bug.




- The programmer tried to force all three processes to be executed for the $\bar{A}\bar{B}\bar{C}$ cases but forgot that the B and C predicates would be done again, thereby bypassing processes A2 and A3.
- Table 6.3 shows the conversion of this flowgraph into a decision table after expansion.

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

***Table 6.3 : Decision Table for Figure 6.5***

**PATH EXPRESSIONS:**

1. **GENERAL:**
   - Logic-based testing is structural testing when it's applied to structure (e.g., control flowgraph of an implementation); it's functional testing when it's applied to a specification.
   - In logic-based testing we focus on the truth values of control flow predicates.
   - A **predicate** is implemented as a process whose outcome is a truth-functional value.
   - For our purpose, logic-based testing is restricted to binary predicates.
   - We start by generating path expressions by path tracing as in Unit V, but this time, our purpose is to convert the path expressions into boolean algebra, using the predicates' truth values (e.g., A and $\overline{A}$) as weights.
2. **BOOLEAN ALGEBRA:**
   - **STEPS:**
     1. Label each decision with an uppercase letter that represents the truth value of the predicate. The YES or TRUE branch is labeled with a letter (say A) and the NO or FALSE branch with the same letter overscored (say $\overline{A}$).
     2. The truth value of a path is the product of the individual labels. Concatenation or products mean "AND". For example, the straight-through path of Figure 6.5, which goes via nodes 3, 6, 7, 8, 10, 11, 12, and 2, has a truth value of ABC. The path via nodes 3, 6, 7, 9 and 2 has a value of $AB\overline{C}$.
     3. If two or more paths merge at a node, the fact is expressed by use of a plus sign (+) which means "OR".

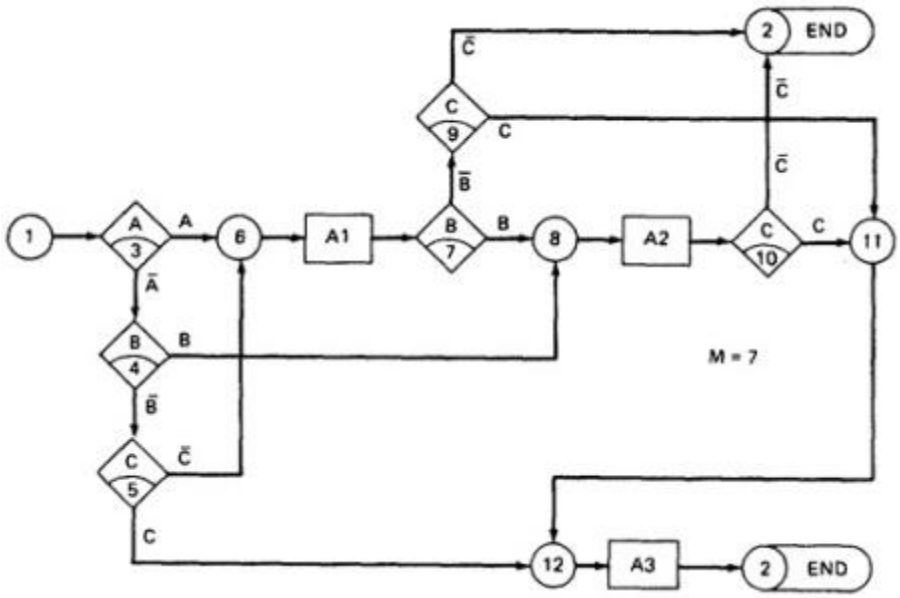



***Figure 6.5 : A Troublesome Program***

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

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

  - There are only two numbers in boolean algebra: zero (0) and one (1). One means "always true" and zero means "always false".
  - **RULES OF BOOLEAN ALGEBRA:**
    - Boolean algebra has three operators: X (AND), + (OR) and $\overline{A}$ (NOT)
    - **X :** meaning AND. Also called multiplication. A statement such as AB (A X B) means "A and B are both true". This symbol is usually left out as in ordinary algebra.
    - **+ :** meaning OR. "A + B" means "either A is true or B is true or both".
    - $\overline{A}$ meaning NOT. Also negation or complementation. This is read as either "not A" or "A bar". The entire expression under the bar is negated.
    - The following are the laws of boolean algebra:

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

- In all of the above, a letter can represent a single sentence or an entire boolean algebra expression.
- Individual letters in a boolean algebra expression are called **Literals** (e.g. A,B)
- The product of several literals is called a **product term** (e.g., ABC, DE).

- An arbitrary boolean expression that has been multiplied out so that it consists of the sum of products (e.g., ABC + DEF + GH) is said to be in **sum-of-products form**.
- The result of simplifications (using the rules above) is again in the sum of product form and each product term in such a simplified version is called a **prime implicant**. For example, ABC + AB + DEF reduces by rule 20 to AB + DEF; that is, AB and DEF are prime implicants.
- The path expressions of Figure 6.5 can now be simplified by applying the rules.
- The following are the laws of boolean algebra

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

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

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

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

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

**KV CHARTS:**

- **INTRODUCTION:**
  - If you had to deal with expressions in four, five, or six variables, you could get bogged down in the algebra and make as many errors in designing test cases as there are bugs in the routine you're testing.
  - **Karnaugh-Veitch chart** reduces boolean algebraic manipulations to graphical trivia.
  - Beyond six variables these diagrams get cumbersome and may not be effective.
- **SINGLE VARIABLE:**
  - Figure 6.6 shows all the boolean functions of a single variable and their equivalent representation as a KV chart.

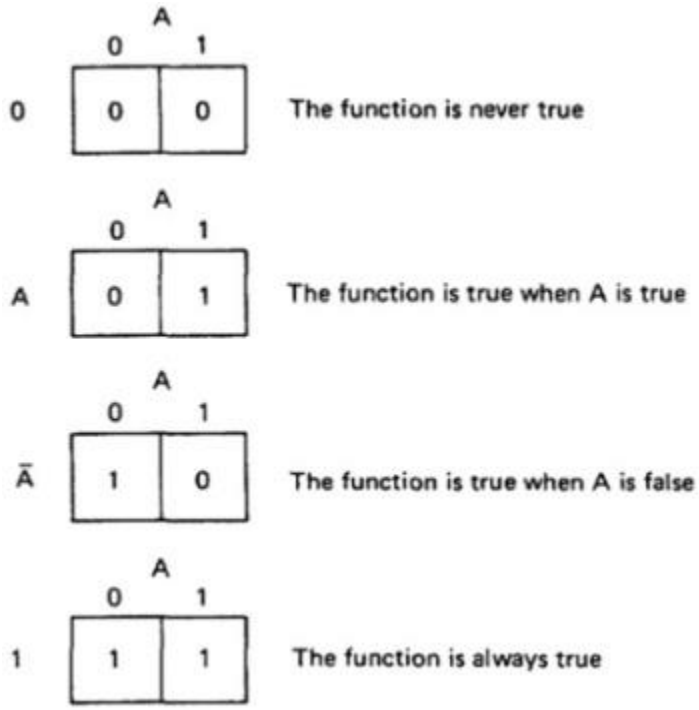



***Figure 6.6 : KV Charts for Functions of a Single Variable.***

**TWO VARIABLES:**

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

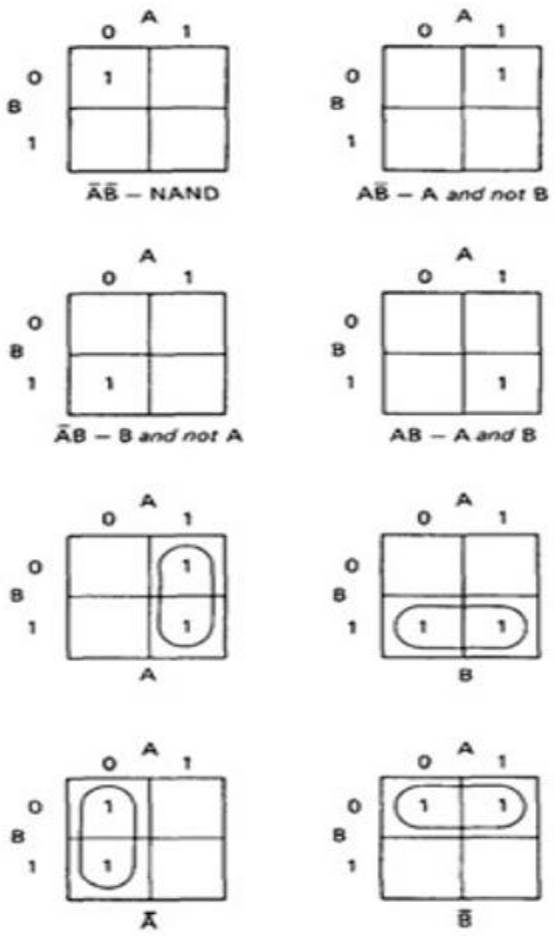



***Figure 6.7 : KV Charts for Functions of Two Variables***

- Each box corresponds to the combination of values of the variables for the row and column of that box.
- A pair may be adjacent either horizontally or vertically but not diagonally.
- Any variable that changes in either the horizontal or vertical direction does not appear in the expression.
- In the fifth chart, the B variable changes from 0 to 1 going down the column, and because the A variable's value for the column is 1, the chart is equivalent to a simple A.

- Figure 6.8 shows the remaining eight functions of two variables.

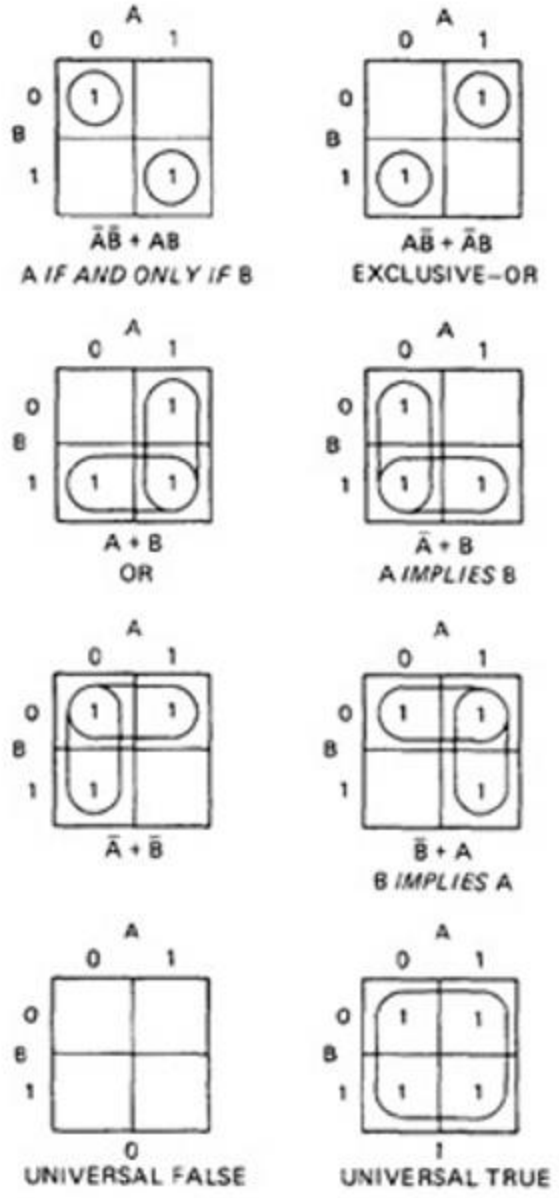
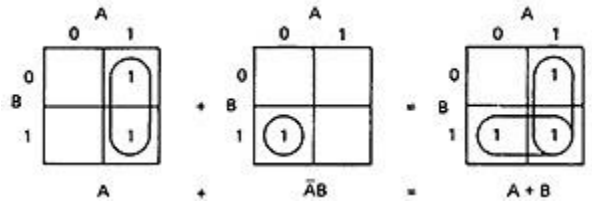



***Figure 6.8 : More Functions of Two Variables.***

- The first chart has two 1's in it, but because they are not adjacent, each must be taken separately.
- They are written using a plus sign.
- It is clear now why there are sixteen functions of two variables.
- Each box in theKV chart corresponds to a combination of the variables' values.
- That combination might or might not be in the function (i.e., the box corresponding to that combination might have a 1 or 0 entry).
- Since n variables lead to $2^n$ combinations of 0 and 1 for the variables, and each such combination (box) can be filled or not filled, leading to $2^{2n}$ ways of doing this.
- Consequently for one variable there are $2^{21} = 4$ functions, 16 functions of 2 variables, 256 functions of 3 variables, 16,384 functions of 4 variables, and so on.

- Given two charts over the same variables, arranged the same way, their product is the term by term product, their sum is the term by term sum, and the negation of a chart is gotten by reversing all the 0 and 1 entries in the chart.




**THREE VARIABLES:**

- KV charts for three variables are shown below.
- As before, each box represents an elementary term of three variables with a bar appearing or not appearing according to whether the row-column heading for that box is 0 or 1.
- A three-variable chart can have groupings of 1, 2, 4, and 8 boxes.

**A few examples will illustrate the principles:**

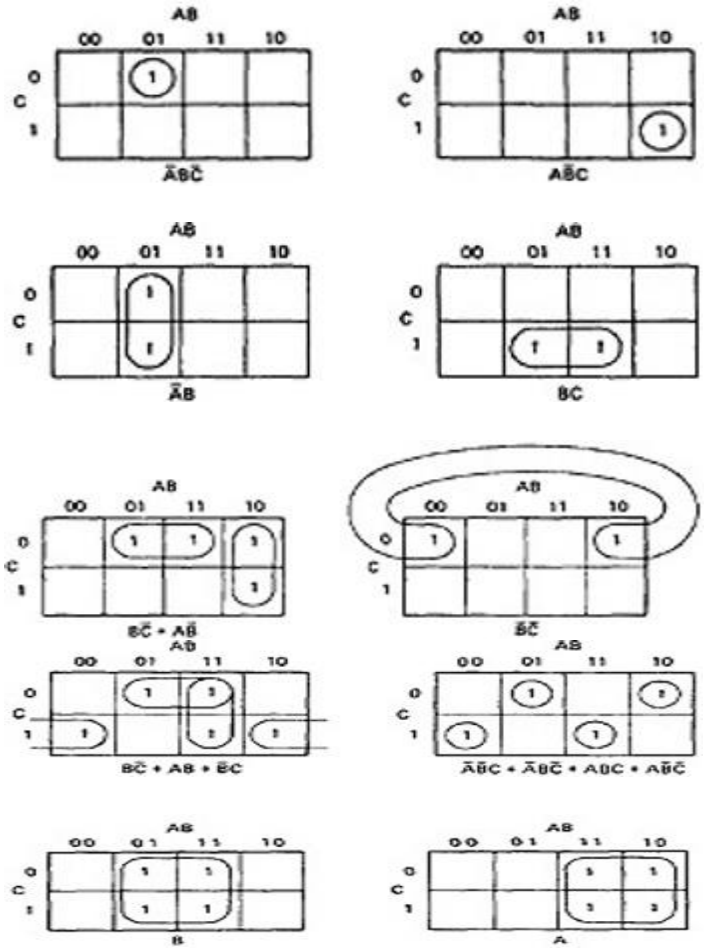
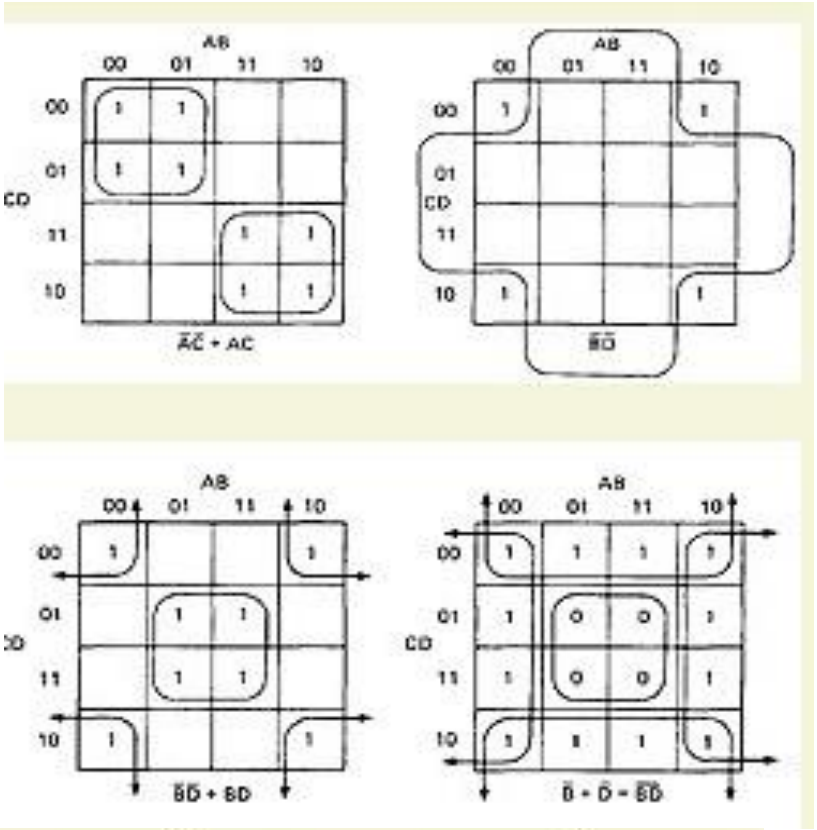
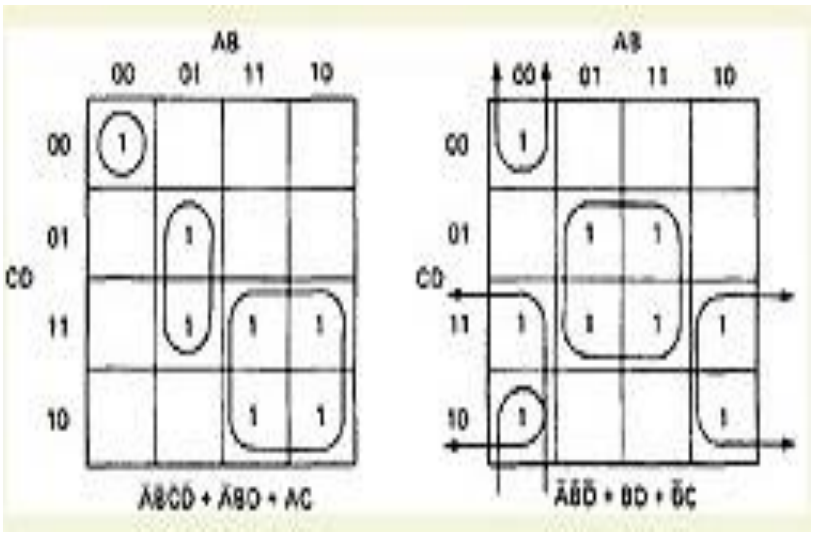



***Figure 6.8 : KV Charts for Functions of Three Variables.***

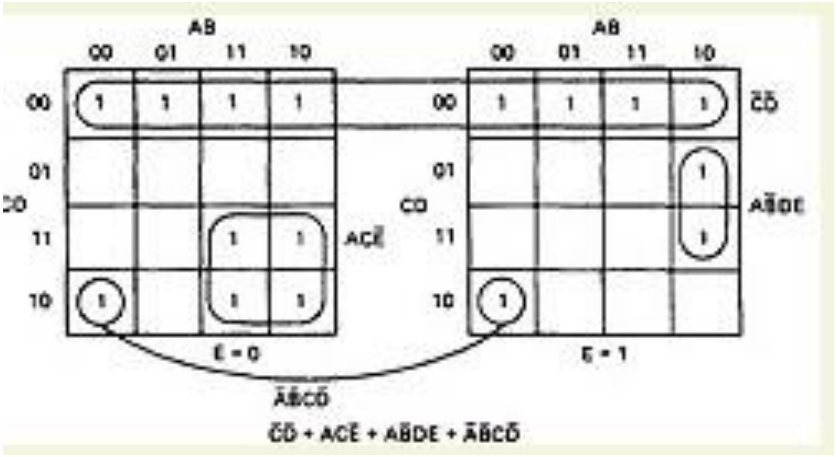
**Four Variables and More**

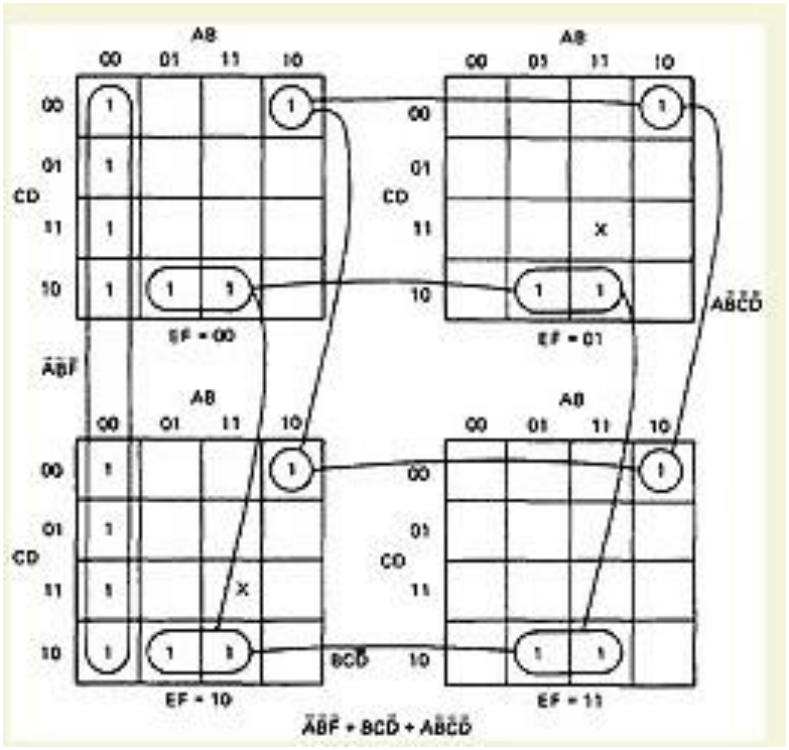
The same principles hold for four and more variables. A four variable chartand several possible adjacencies are shown below. Adjacencies can now consists of 1,2,4,8 and 16 boxes




- A five & Six variable KV chart is shown below


AB
00 01 11 10
CD
E = 0
E = 1
ACE
ABDE
CD
ABCD
CD + ACE + ABDE + ABCD


AB
00 01 11 10
CD
EF = 00
EF = 01
EF = 10
EF = 11
ABF
BCD
ABCD
ABF + BCD + ABCD

**Even More Testing Strategies?**

**1.** Use one prime implicant per domain. You'll obviously miss parts of the domain notcovered by the prime implicant you chose—e.g., for disconnected domains.

**2.** Use the prime implicant with the fewest variables and work down to the prime implicant with the greatest number of variables

**3.** Overcome the obvious weaknesses of the above strategy (not all subdomains are covered)by using one test per prime implicant.

**4.** Every term in the product form for *n* variables has at most *n* literals but, because of simplifications made possible by adjacencies, terms may have fewer than *n* literals, say *k*.Any term with *k* literals can be expanded into two terms with *k* + 1 literals

## 4.1 SPECIFICATIONS

### 4.4.1. General

The procedure for specification validation is straightforward:

**1.** Rewrite the specification using consistent terminology.

**2.** Identify the predicates on which the cases are based. Name them with suitable letters,such as A, B, C.

**3.** Rewrite the specification in English that uses only the logical connectives AND, OR, andNOT, however stilted it might seem.

**4.** Convert the rewritten specification into an equivalent set of boolean expressions

**5.** Identify the default action and cases, if any are specified.

**6.** Enter the boolean expressions in a KV chart and check for consistency. If the specifications are consistent, there will be no overlaps, except for cases that result in multipleactions.

**7.** Enter the default cases and check for consistency.

**8.** If all boxes are covered, the specification is complete.

**9.** If the specification is incomplete or inconsistent, translate the corresponding boxes of theKV chart back into English and get a clarification, explanation, or revision.

**10.** If the default cases were not specified explicitly, translate the default cases back intoEnglish and get a confirmation.

### 4.4.2. Finding and Translating the Logic

The specifications into sentences of the following form:

–IF predicate THEN action.‖

**If**—based on, based upon, because, but, if, if and when, only if, only when, provided that,when, when or if, whenever.

**Then**—applies to, assign, consequently, do, implies that, infers that, initiate, means that,shall, should, then, will, would.

**And**—all, and, as well as, both, but, in conjunction with, coincidental with, consisting of, comprising, either . . . or, furthermore, in addition to, including, jointly, moreover, mutually,plus, together with, total, with.

**OR**—and, and if . . . then, and/or, alternatively, any of, anyone of, as well as, but, case, contrast, depending upon, each, either, either . . . or, except if, conversely, failing that, furthermore, in addition to, nor, not only . . . but, although, other than, otherwise, or, or else,on the other hand, plus.

**NOT**—but, but not, by contrast, besides, contrary, conversely, contrast, except if, excluding,excepting, fail, failing, less, neither, never, no, not, other than.

**EXCLUSIVE OR**—but, by contrast, conversely, nor, on the other hand, other than, or.

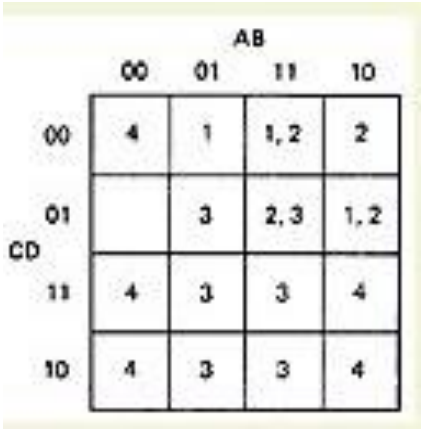
**IMMATERIAL**—independent of, irregardless, irrespective, irrelevant, regardless, but not if,whether or not.

### 4.4.3. Ambiguities and Contradictions

* Specification is

$$
\begin{aligned}
A1 &= B\bar{C}\bar{D} + A\bar{B}\bar{C}D \\
A2 &= A\bar{C}\bar{D} + A\bar{C}D + A\bar{B}\bar{C} + AB\bar{C} \\
A3 &= BD + B\bar{C}\bar{D} \\
ELSE &= B\bar{C} + \bar{A}B\bar{C}\bar{D}
\end{aligned}
$$

* Here is the KV chart for this specification (I've used the numerals 1, 2, 3, and 4 torepresent the actions and the default case):




There is an ambiguity probably related to the default case : $\bar{A}B\bar{c}\bar{D}$ is missing

When the specifier asks about his ambiguity the end users answers are contradictions.

It is necessary to resolve those ambiguities by presenting the complete specification in form oftables to the end user

**4.4 4. Don't-Care and Impossible Terms**

There are only three things in this universe that certain are impossible:

1. Solving a provably unsolvable problem, such as creating a universal program verifier.
2. Knowing both the exact position and the exact momentum of a fundamental particle.
3. Knowing what happened before the ‑big bang‖ that started the universe. There are two

kinds of so-called impossible conditions:

(1) the condition cannot be created or is seemingly contradictory or improbable;

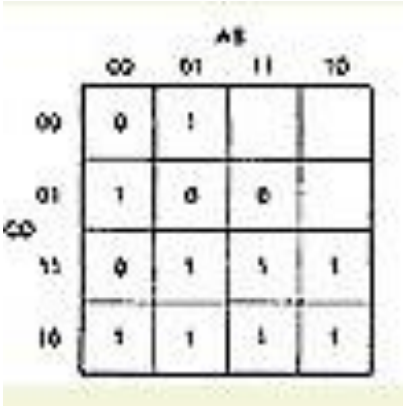
(2) the condition results from our insistence on forcing a complex, continuous world into abinary, logical mold.

- Most program illogical conditions are of the latter kind. There are twelve cases forsomething, say, and we represent those cases by 4 bits
- Our conversion from the continuous world to binary world
- Taking advantage of impossible conditions is a dangerous practice and should be avoidedbut if you are insisted on doing that sort of the thing we must to do it right so the steps areas follows

1. Identify all ‑impossible‖ and ‑illogica1‖ cases and confirm them.
2. Document the fact that you intend to take advantage of them.

3. Fill out the KV chart with the possible cases and then fill in the impossible cases. Use the combined symbol φ, which is to be interpreted as a 0 or 1, depending on which value provides the greatest simplification of the logic. These terms are called don't-care terms, because the case is presumed impossible, and we don't care which value (0 or 1) is used.

4. Here is an example:



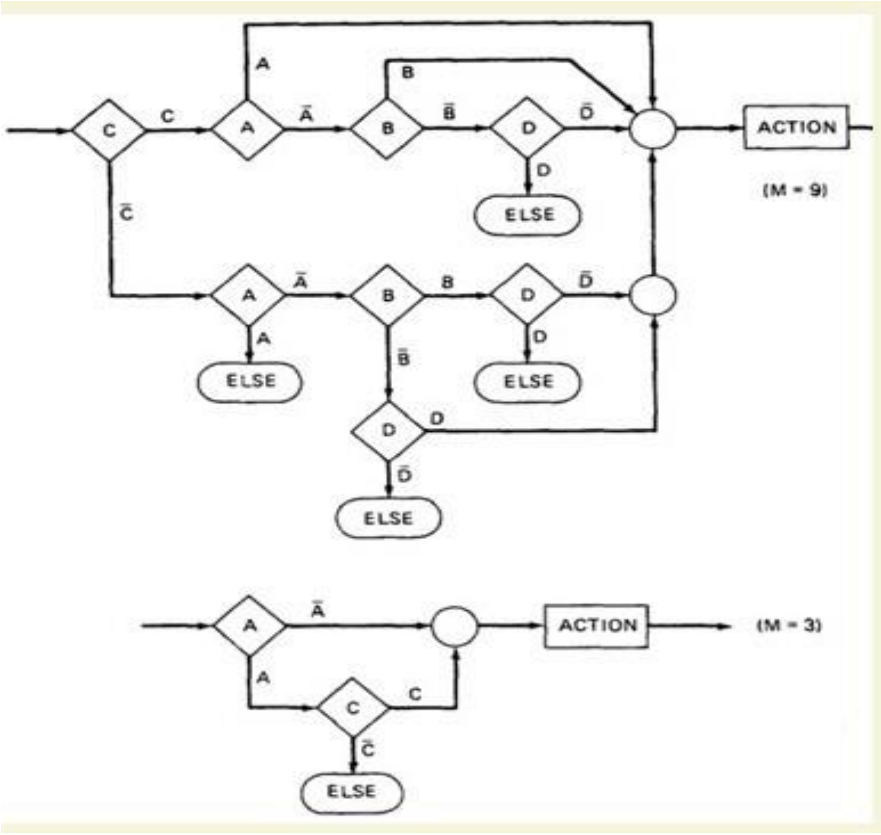
By not taking the advantage of the impossible conditions, we get the resulting Boolean expression

$C\overline{D} + CB + CA + \overline{A}B\overline{D} + \overline{A}\overline{B}\overline{C}D$

* By taking advantage of the impossible conditions, we get:

$C + \overline{A}$

The corresponding flow graph as shown below


A
B
C
C
A
Ā
B
B̄
D
D̄
ACTION
D
(M = 9)
ELSE
C̄
A
Ā
B
B
D
D̄
A
D
ELSE
B̄
ELSE
D
D
D̄
ELSE
A
Ā
ACTION
(M = 3)
A
C
C
C̄
ELSE

**UNIT-V**

**Objective of the unit:**

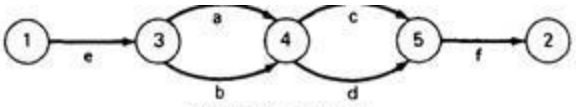
Students learn about basics of Path, with the help of reduction procedure calculate maximum path count and mean processing time andprobability of a particular point along the routine and also know all the basics of graph matrices.

***Topic:*** **PATHS, PATH PRODUCTS AND REGULAR EXPRESSIONS, GRAPH MATRICES**

***Motivation (Why this topic is significant for the discussion?):***

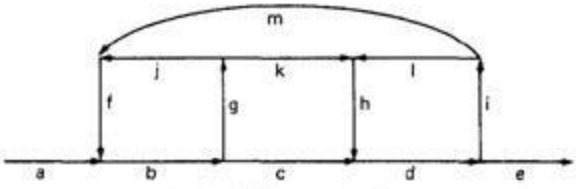
- Flow graphs are being an abstract representation of programs.
- Any question about a program can be cast into an equivalentquestion about an appropriate flowgraph.
- Most software development, testing and debugging tools useflow graphs analysis techniques.

- ***Notes:*** **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 whichdenotes 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.
  - The name of the path or path segment that corresponds to those links is expressed naturally by concatenating those linknames.
  - 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.



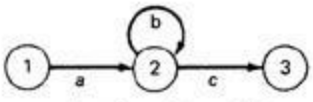
eacf, eadf, ebcf, ebdf

(a)



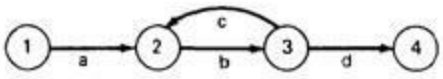
abcde, abgjfbcde, abcdimfbcde

(b)



ac, abc, abbc, abbbe, abbbbc

(c)



abd, abcbd, abcbcbd, abcbcbcbd

(d)

- **PATH EXPRESSION:**
  - Consider a pair of nodes in a graph and the set of paths between those node.

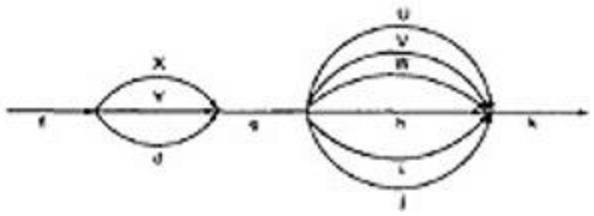
    Denote that set of paths by Upper case letter such as X,Y

    The + sign is understood to mean "or" between the two nodesof interest.

- **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

- **PATH SUMS:**
  - The "+" sign was used to denote the fact that path nameswere part of the same set of paths.
  - The "PATH SUM" denotes paths in parallel between nodes.



The first set of parallel paths is denoted by X + Y + d and thesecond set by U + V + W + h + i + j.

- 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

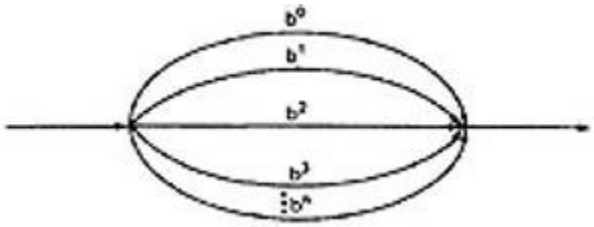
    RULE 4: A(B+C)=AB+AC and (B+C)D=BD+CD

  - 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)

- **LOOPS:**

  Loops can be understood as an infinite set of parallel paths.



- **REDUCTION PROCEDURE ALGORITHM:**
  - This section presents a reduction procedure for converting a flow graph whose links are labeled with names into a path expression that denotes the set of all entry/exit paths in that

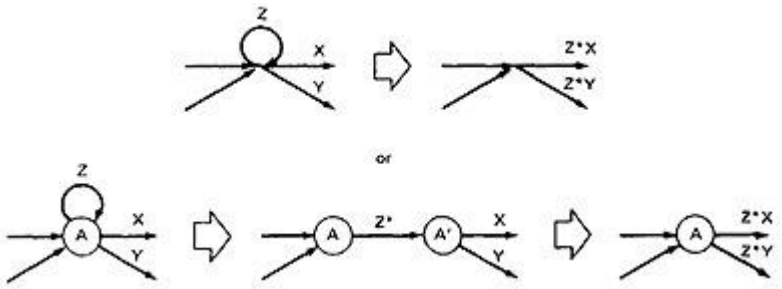
flow graph. The procedure is a node-by-node removalalgorithm.

- 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) byreplacing 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.

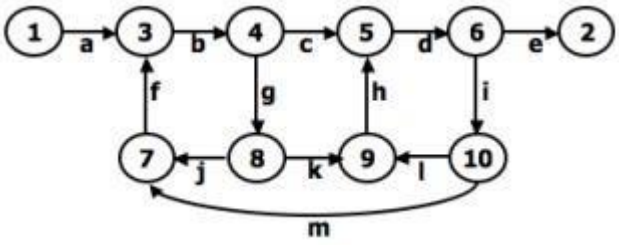
- **LOOP REMOVAL OPERATIONS:**
  - There are two ways of looking at the loop-removal operation:



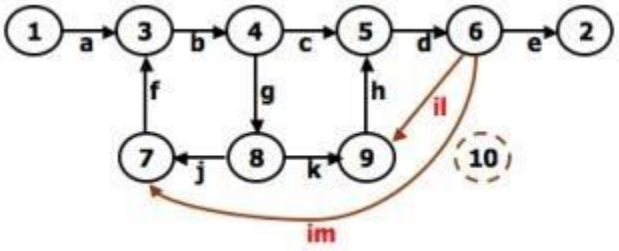

  - In the first way, we remove the self-loop and then multiply all outgoing links by Z*.
  - In the second way, we split the node into two equivalent nodes, call them A and A' and put in a link between them whose path expression is Z*. Then we remove node A' using steps 4 and 5 to yield outgoing links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

- **A REDUCTION PROCEDURE - EXAMPLE:**

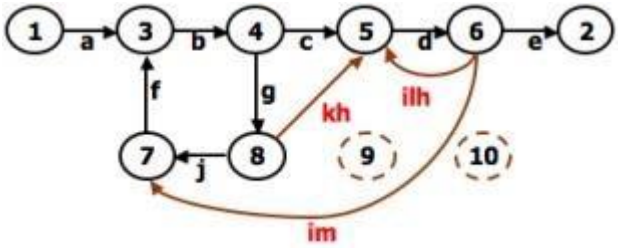
- Let us see by applying this algorithm to the following graphwhere we remove several nodes in order; that is



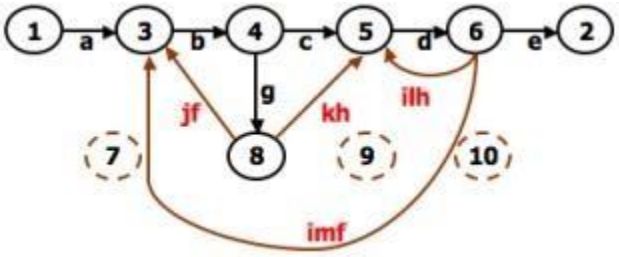

- Remove node 10 by applying step 4 and combine by step 5 toyield



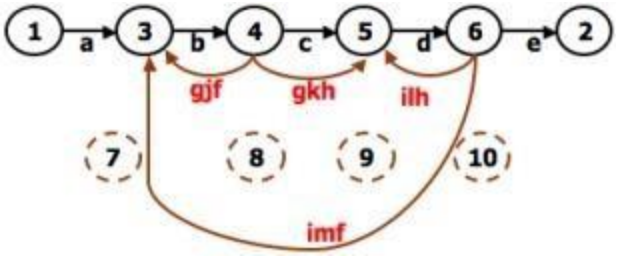

- Remove node 9 by applying step4 and 5 to yield



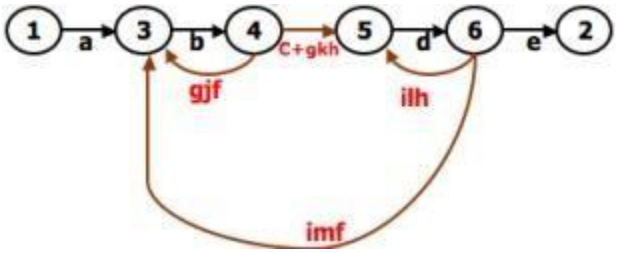

- Remove node 7 by steps 4 and 5, as follows:



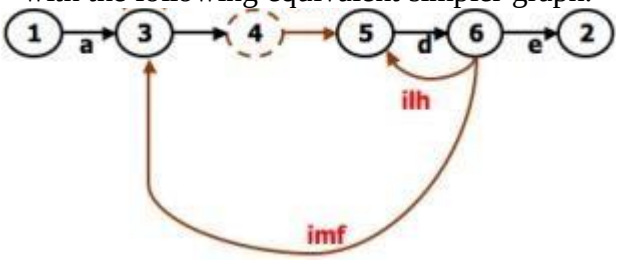

- Remove node 8 by steps 4 and 5, to obtain:




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

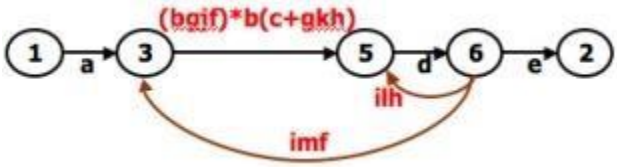



- **LOOPTERMSTEP**
  Removing node 4 leads to a loop term. The graph has now been replaced with the following equivalent simpler graph:

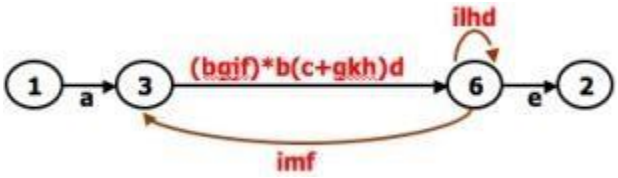



o

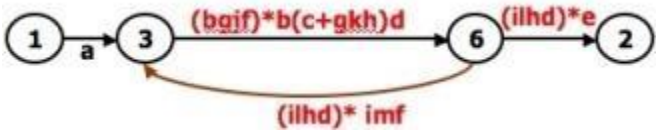
- Continue the process by applying the loop-removal step as follows:



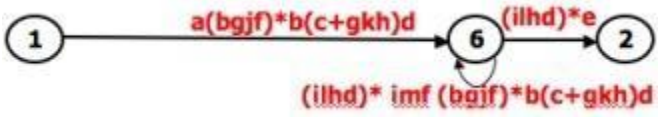

- Removing node 5 produces:




- 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

- **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.

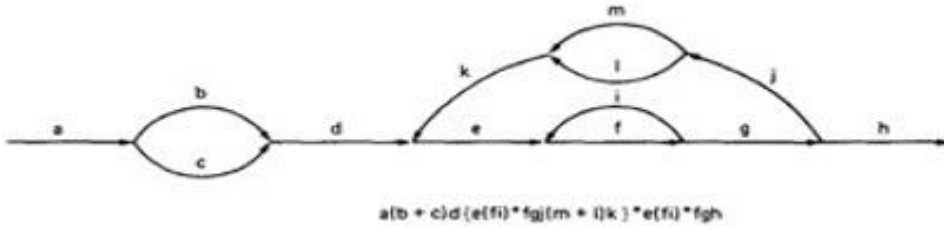
**MAXIMUM PATH COUNT ARITHMETIC:**

- 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:**
  - The following is a reasonably well-structured program.

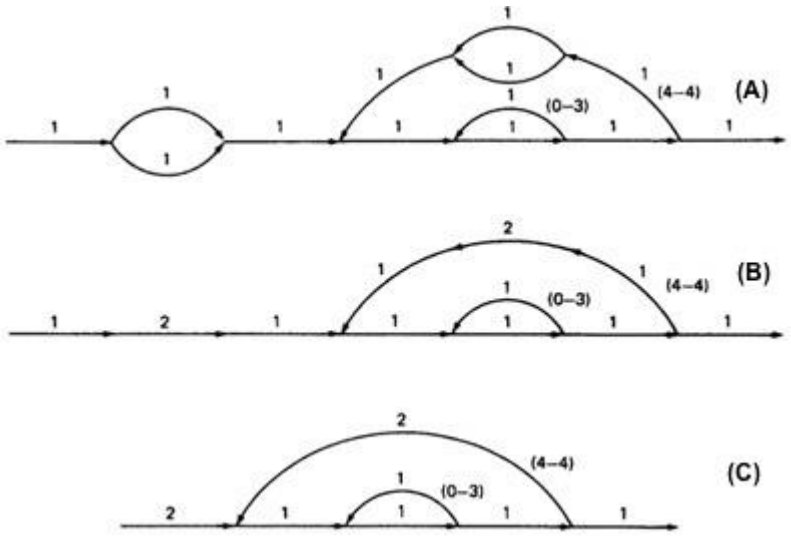



a(b + c)d {e(fi)*fgj(m + l)k }*e(fi)*fgh

- 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

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

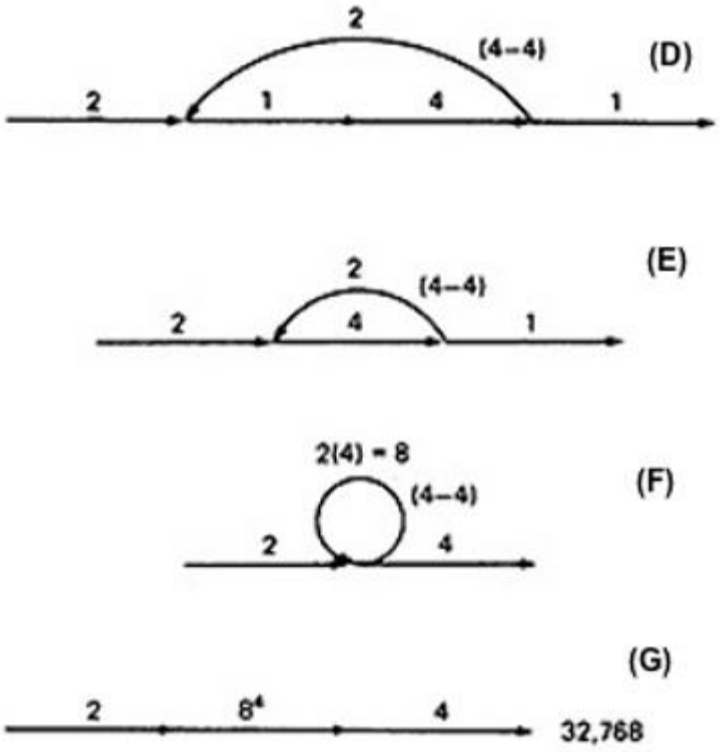



- **FortheInnerLoop:**
  **D:**Calculate the total weight of inner loop, which can execute a min. of 0 times and max. of 3 times. So, itinnerloopcanbeevaluatedasfollows:

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

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

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




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

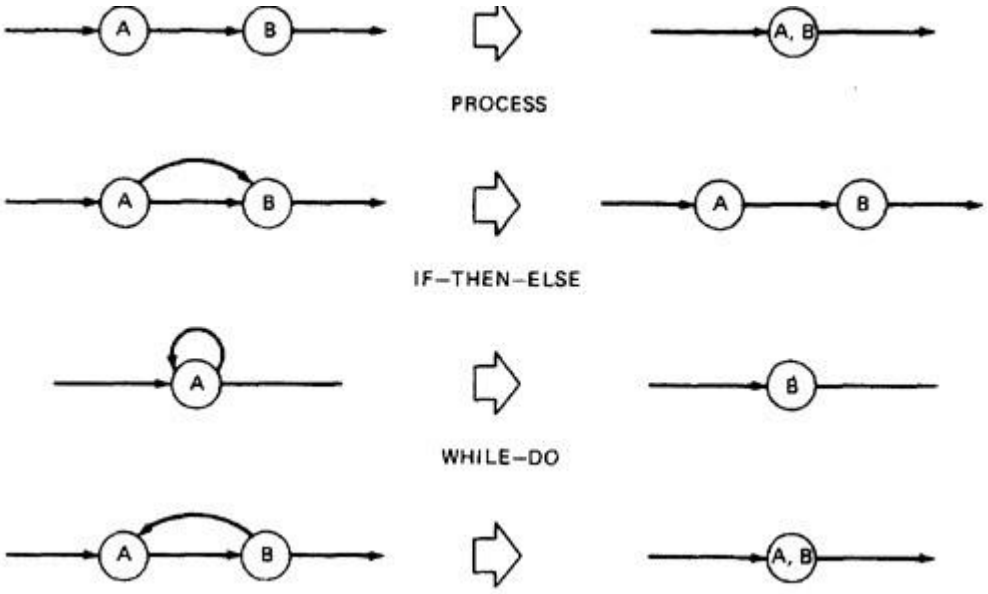
  **a(b+c)d{e(fi)*fgj(m+l)k}*e(fi)*fgh**
  = $1(1 + 1)1(1(1 \times 1)^{\underline{3}}1 \times 1 \times 1(1 + 1)1)^{\underline{4}}1(1 \times 1)^{\underline{3}}1 \times 1 \times 1$
  $=2(1^{\underline{3}}1x(2))^4 1^{\underline{3}}$

- $=2(4x2)^4 \times 4$
  $= 2 \times 8^4 \times 4$

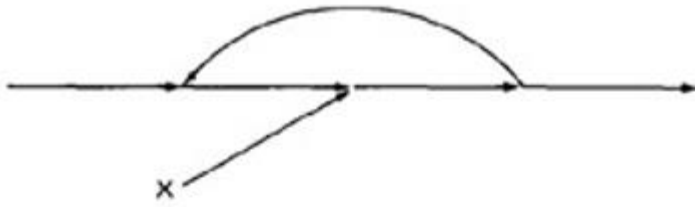
- $= 32,768$

- 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.

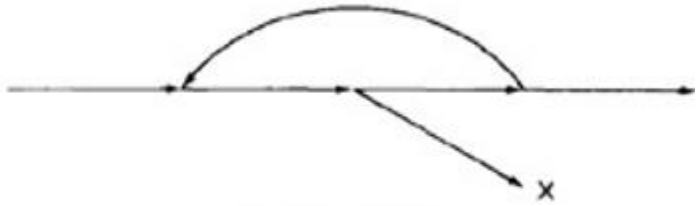
- **STRUCTURED FLOWGRAPH:**
  - Structured code can be defined in several different ways that do not involve ad-hoc rules such as not using GOTOs.
  - A structured flowgraph is one that can be reduced to a single link by successive application of the transformations



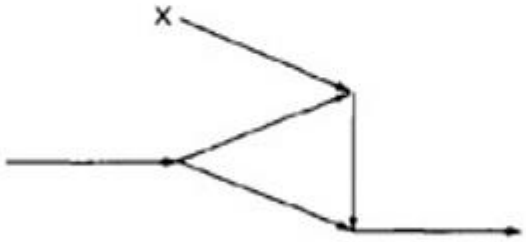

- he node-by-node reduction procedure can also be used as a test for structured code.
- Flow graphs that DO NOT contain one or more of the graphs shown below (Figure 5.8) as subgraphs are structured.
    1. Jumping into loops
    2. Jumping out of loops
    3. Branching into decisions
    4. Branching out of decisions



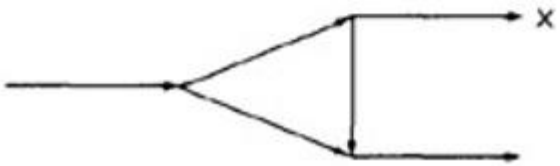

Jumping Into Loops




Jumping Out of Loops




Branching Into Decisions



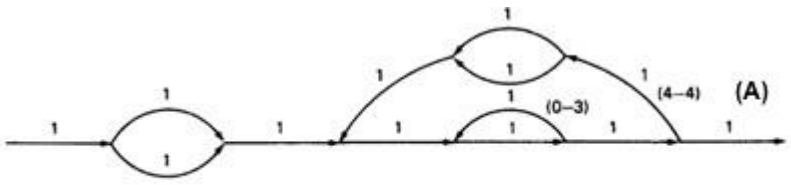

Branching Out of Decisions

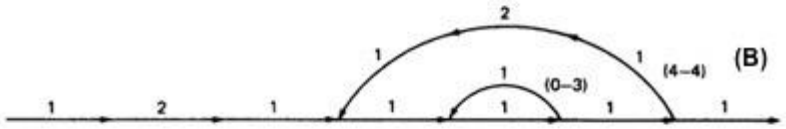
**LOWER PATH COUNT ARITHMETIC:**

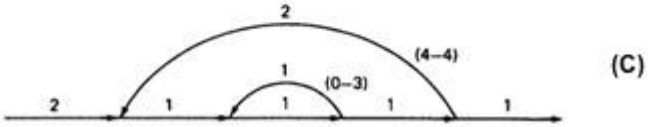
- A lower bound on the number of paths in a routine can be approximated for structured flow graphs.
- The arithmetic is as follows:

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

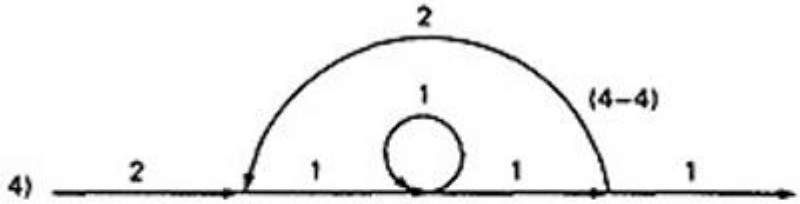
- The values of the weights are the number of members in a set of paths.
- **EXAMPLE:**
  - Applying the arithmetic to the earlier example gives us the identical steps unitl step 3 (C) as below:

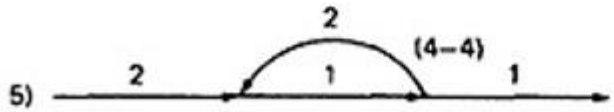


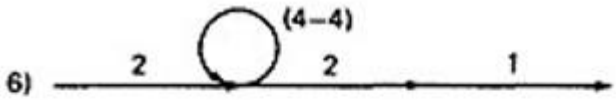



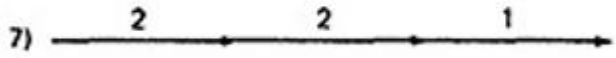



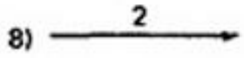

From Step 4, the it would be different from the previous example:



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

**CALCULATING THE PROBABILITY:**

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

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

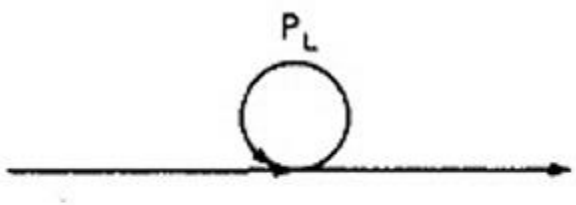
  This question can be answered under suitable assumptions, primarily that all probabilities involved are independent, which is to say that all decisions are independent and uncorrelated.
- We use the same algorithm as before : node-by-node removal of uninteresting nodes.
- **Weights, Notations and Arithmetic:**
  - Probabilities can come into the act only at decisions (including decisions associated with loops).
  - Annotate each outlink with a weight equal to the probability of going in that direction.
  - Evidently, the sum of the outlink probabilities must equal 1
  - For a simple loop, if the loop will be taken a mean of N times, the looping probability is N/(N + 1) and the probability of not looping is 1/(N + 1).
  - A link that is not part of a decision node has a probability of 1.
  - The arithmetic rules are those of ordinary arithmetic.

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

- In this table, in case of a loop, $P_A$ is the probability of the link leaving the loop and $P_L$ is the probability of looping.
- 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.

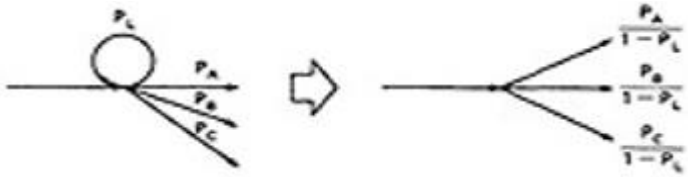
- For example, a loop node has a looping probability of $P_L$ and a probability of not looping of $P_A$, which is obviously equal to I - PL




$$P_A = 1 - P_L$$

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

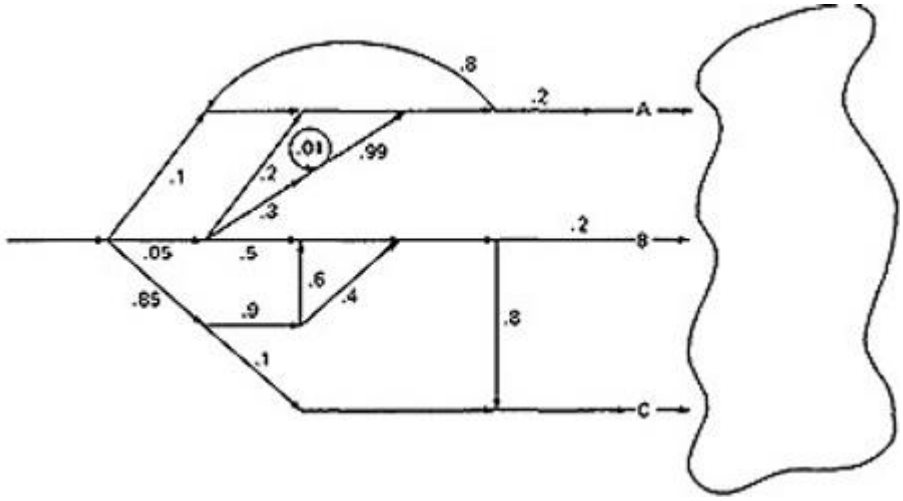
Following the above rule, all we've done is replace the outgoing probability with 1 - so why the complicated rule? After a few steps in which you've removed nodes, combined parallel terms, removed loops and the like, you might find something like this:



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

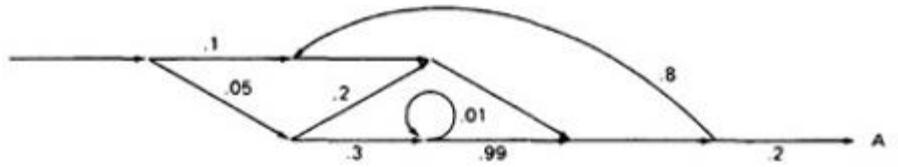
$$\frac{P_A}{1 - P_L} + \frac{P_B}{1 - P_L} + \frac{P_C}{1 - P_L} = \frac{P_A + P_B + P_C}{1 - P_L} = 1$$

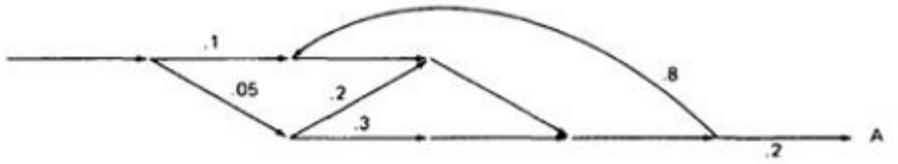
- 
  - which is what we've postulated for any decision. In other words, division by 1 - $P_L$ renormalizes the outlink probabilities so that their sum equals unity after the loop is removed.
- **EXAMPLE:**
  - Here is a complicated bit of logic. We want to know the probability associated with cases A, B, and C.

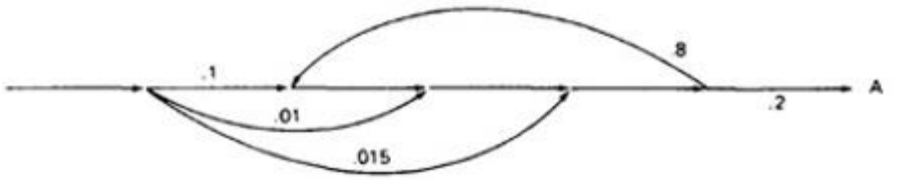


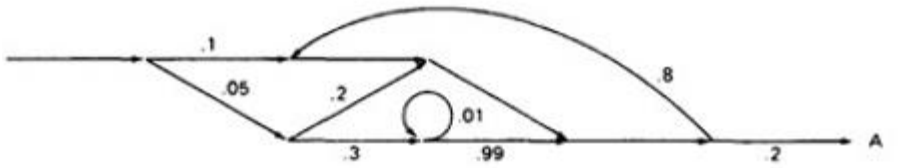

Let us do this in three parts, starting with case A. Note that the sum of the probabilities at each decision node is equal to 1. Start by throwing away anything that isn't on the way to case A, and then apply the reduction procedure. To avoid clutter, we usually leave out probabilities equal to 1.

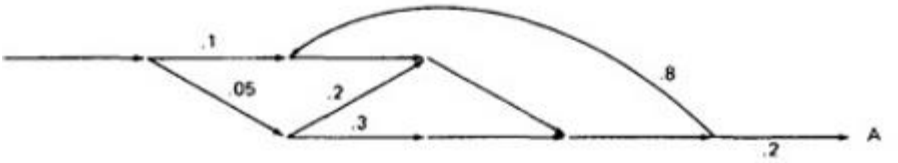
CASE A:

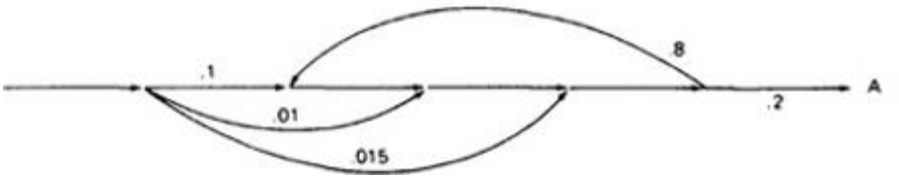


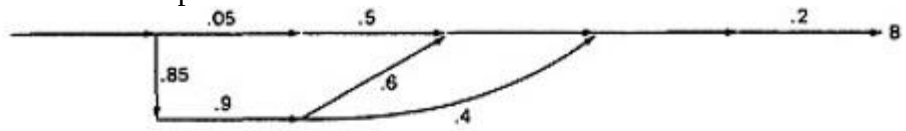



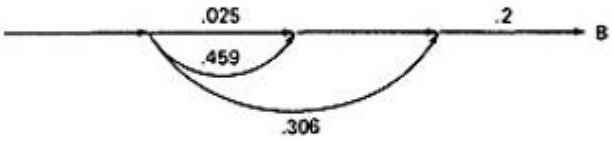



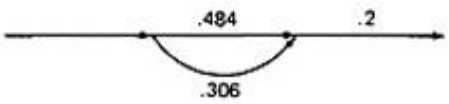



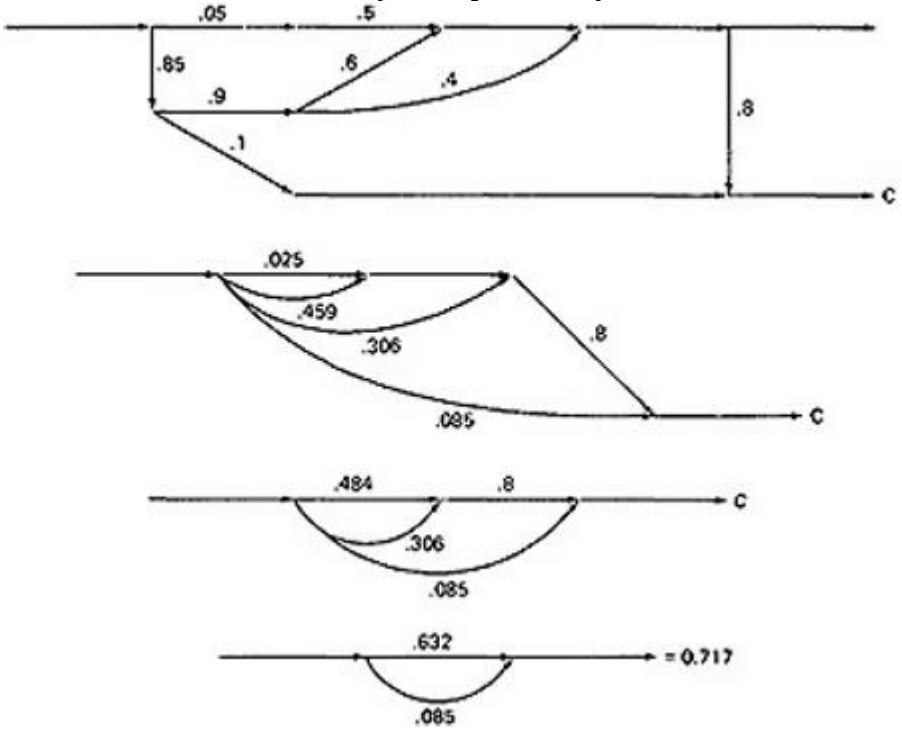

Case B is simpler




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



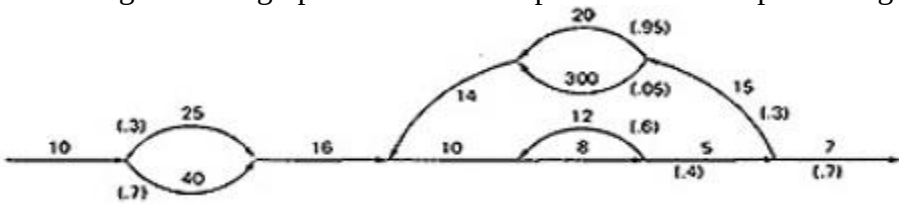

- This checks. It's a good idea when doing this sort of thing to calculate all the probabilities and to verify that the sum of the routine's exit probabilities does equal 1.
- If it doesn't, then you've made calculation error or, more likely, you've left out some branching probability.
- How about path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

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

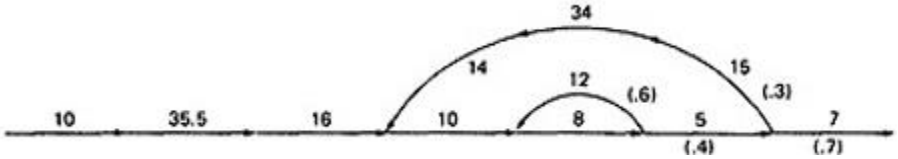
| Case | Path expression | Weight expression |
|---|---|---|
| Parallel | A+B | $T_{A+B}=(P_AT_A+P_BT_B)/(P_A+P_B)$<br>$P_{A+B}= P_A+P_B$ |
| Series | AB | $T_{AB}=T_A+T_B$<br>$P_{AB} =P_AP_B$ |
| Loop | $A^n$ | $T_A= T_A+ T_LP_L/(1-P_L)$<br>$P_A =P_A/(1-P_L)$ |

**EXAMPLE:**

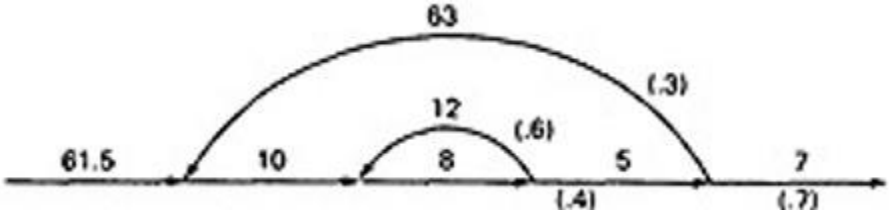
- Start with the original flow graph annotated with probabilities and processing time



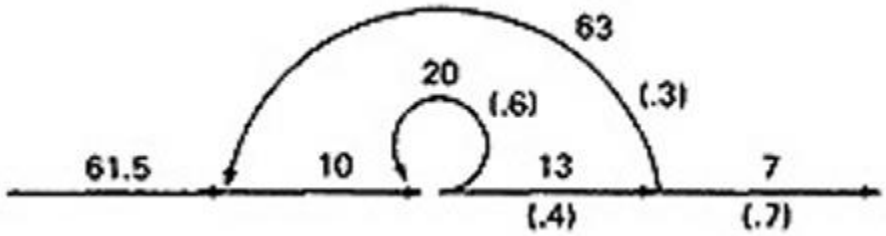

- Combine the parallel links of the outer loop. The result is just the mean of the processing times for the links because there aren't any other links leaving the first node. Also combine the pair of links at the beginning of the flowgraph..



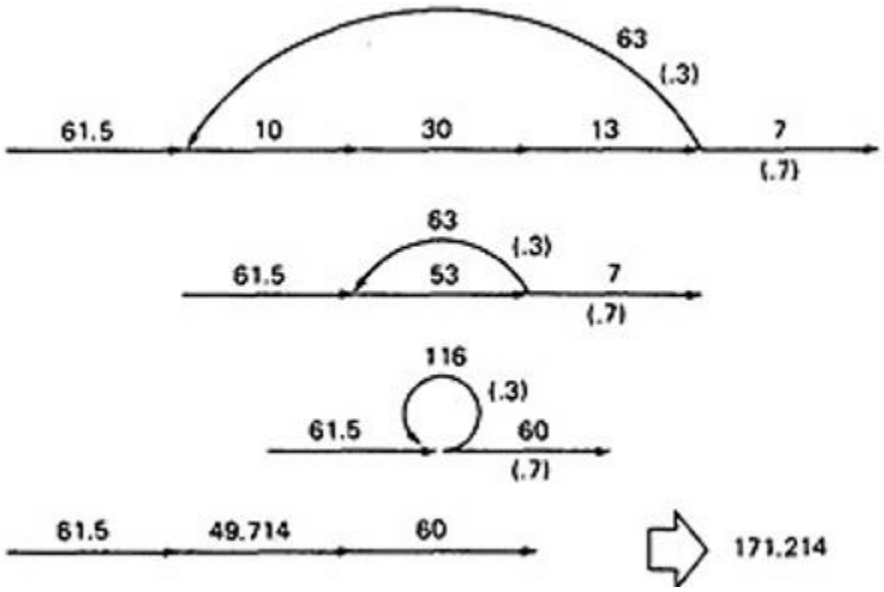

- Combine as many serial links as you can




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




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




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

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

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

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

- **EXAMPLE 1 (PUSH / POP):**
  - Here is the Push/Pop Arithmetic:

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

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

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

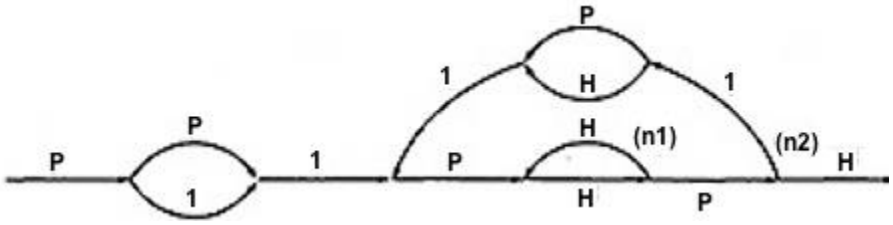
PUSH/POP MULTIPLICATION TABLE

| X | H PUSH | P POP | 1 NONE |
|---|---|---|---|
| H | $H^2$ | 1 | H |
| P | 1 | $P^2$ | P |
| 1 | H | P | 1 |

PUSH/POP ADDITION TABLE

| + | H PUSH | P POP | 1 NONE |
|---|---|---|---|
| H | H | P + H | H + 1 |
| P | P + H | P | P + 1 |
| 1 | H + 1 | P + 1 | 1 |

Consider the following flowgraph




- $P(P + 1)1\{P(HH)^{n1}HP1(P + H)1\}^{n2}P(HH)^{n1}HPH$
- Simplifying by using the arithmetic tables,
- $=(P^2 + P)\{P(HH)^{n1}(P + H)\}^{n1}(HH)^{n1}$
- $=(P^2 + P)\{H^{2n1}(P^2 + 1)\}^{n2}H^{2n1}$
- Below Table 5.9 shows several combinations of values for the 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 | $\sum_{8}^{15} H^i$ |

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

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

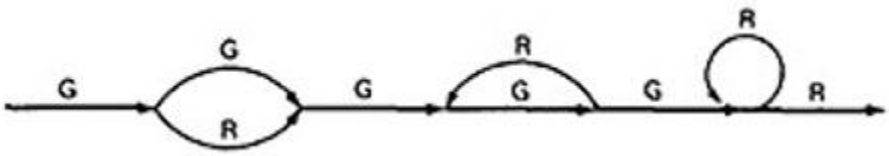
**Multiplication Table**

| X | G | R | 1 |
|---|---|---|---|
| G | $G^2$ | 1 | G |
| R | 1 | $R^2$ | R |
| 1 | G | R | 1 |

**Addition Table**

| + | G | R | 1 |
|---|---|---|---|
| G | G | G + R | G + 1 |
| R | G + R | R | R + 1 |
| 1 | G + 1 | R + 1 | 1 |

- G" denotes GET and "R" denotes RETURN.
- Consider the following flowgraph:




- G(G + R)G(GR)*GGR*R
- = G(G + R)$G^3$R*R
- = (G + R)$G^3$R*
- = ($G^4$ + $G^2$)R*
- This expression specifies the conditions under which the resources will be balanced on leaving the routine.
- If the upper branch is taken at the first decision, the second loop must be taken four times.
- If the lower branch is taken at the first decision, the second loop must be taken twice.
- For any other values, the routine will not balance. Therefore, the first loop does not have to be instrumented to verify this behavior because its impact should be nil.

**LIMITATIONS AND SOLUTIONS:**

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

***REGULAR* EXPRESSIONS *AND FLOW ANOMALY DETECTION:***

- **THE PROBLEM:**
  - The generic flow-anomaly detection problem (note: not just data-flow anomalies, but any flow anomaly) is that of looking for a specific sequence of options considering all possiblepaths 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 requiresus to detect the *dd*, *dk*, *kk*, and *ku* sequences. Are there paths with anomalous data flows?

**THE METHOD:**

- Annotate each link in the graph with the appropriate operator or the null operator 1.
- Simplify things to the extent possible, using the fact that a + a = a and 12 = 1.
- 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.

**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 uried in an expression.

**Unit-VI**

**Graph matrices and pplications**

**Motivation:**

Graphs are introduced as software structure, in graphs path tracing is easy if there are many number of paths there is confused in tracing paths.

One solution to this problem is to represent the graphs as a matrix and to usematrix operations equivalent to path tracing

**The basic algorithm**

1. Matrix multiplication, which is used to get path expression from every node to other
2. A partitioning algorithm is used to convert loops into loopfree
3. A collapsing process which gets the path expression fromany node to any another node

**The matrix of a graph**

**Basic principle**

A Graph matrix is a square array with one row and column for every node inthe graph. Each row column combination shows a relation.

Need to observe some of the points regarding this

1. The size of the matrix equals the no. of nodes
2. There is a possible to put every possible direct connection or link between any node to any other node
3. The entry at a row and column intersection is the link weight of the link that connects two nodes in that direction
4. A connection from node i to node j does not imply a connection from node j to node i.
5. If there are several links between two nodes then the entry is sum, the "+" sign denotes parallel links as usual.

**A simple weight**

- If there is a connection indicates by "1" not "0".
- A matrix with weight as 1 and 0 then it is called a connection matrix
- Each row of a matrix represents the outlink to that node and eachcolumn represents inlink to corresponding node.
- A branch node is a node with more than one non zero entry. A junctionnode is a node

more than one non-zero entry in it column.

**Relations**

- A graph consists of a set of abstract objects called nodes and relation R between the nodes. If aRb, which is to say that a has a relation R to b, it is denoted by a link from a to b.links have link weights , link weights may be logical, illogical, objective, subjective .
- Graphs defined over " is connected to" are called connectin matrices
- **Properties of relations**

**Transitive relations**

- A relation R is transitive if aRb and bRc implies aRc.
- Ex: is connected to , is grater than , is less than , is greater than or equalto, is slower than, is faster than

**Reflexive relation**

- A relation R is reflexive if, for every a and b, aRb implies bRa.
- Means if there is a link from a to b then there is a also link from b to a
- A graph whose relation is not symmetric is called a directed graph notthen it is undirected graph.

**Antisymmetric relation**

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

**Equivalent relations**

It is a relation that satisfies the reflexive, ransitive , and symmetricproperties.

**Partial ordering Relations**

This relation satisfies the reflexive, transitive and antisymmetric properties. Partial ordering relations having some important properties they are loop free, there is atleast one maximum element, there is atleast one minimum element,

A **maximum** element is one for which the relation xRa does not hold for any other element x.

A **Minimum element** a is for which the relation aRx does not holdfor any other element x.

**The power of a matrix**

a. **Principle**

In graph matrices an expression states as

1. Consider the relation between every node and its neighbor

2. Extend the relation by considering each neighbor as anintermediate node
3. At Extend further by considering each neighbor's neighbor as an intermediate node
4. Continue until the longest possible no repeating path has been established.
5. Do this for every pair of nodes in the graph.

**Matrix powers and products**

The square of the matrix obtained by replacing every entry with aij= ∑aikakj(sigma is form 1 to n)

$A^2A=AA^2$ that is matrix multiplication is associative.

- A matrix for which $A^2=A$ is said to be Idempotent. A matrix whose successive powers eventually yield an idempotent matrix is called an idempotent generator.
- The nth power of a matrix A+I over a transitive relation is called the **transitive closure** of the matrix.

**Loops**

- Every loop forces us into potentially sum of matrix powers.
- In a graph matrices entries with diagonal are self loops.
- Predicate loops come about form declared or undeclared programswitches and.or unstructured loop constructs.

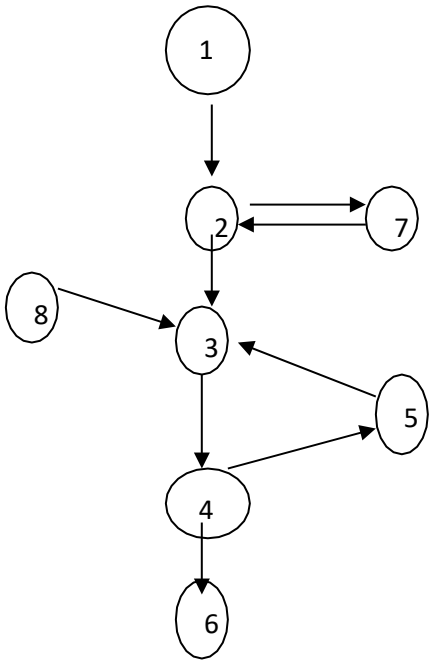
**Partitioning algorithm**

Partition the graph by grouping nodes in such way thatevery loop within onegroup or another. Such a graph is partly ordered.

There are many used for an algorithm that does that as

1. Embed te loops within a subroutine so as to have a resulting graph which is loop-free at the top level.
2. Graphs with loops are easy to analyze if know where tobreak.
3. Recognize loops is much harder to do it unless need solid algorithm on which to base the tool.

To achieve partitioning algorithm have an expression such as$(A+I)^n+\#(A+I)^{nT}$ .




The relation matrix is

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | | | | | | |
| | 1 | 1 | | | | 1 | |
| | | 1 | 1 | | | | |
| | | | 1 | 1 | 1 | | |
| | | 1 | | 1 | | | |
| | | | | | 1 | | |
| | 1 | 1 | | | | 1 | |
| | | 1 | | | | | 1 |

The transitive closure matrix is

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| | 1 | 1 | 1 | 1 | 1 | 1 | |
| | | 1 | 1 | 1 | 1 | | |
| | | 1 | 1 | 1 | 1 | | |
| | | 1 | 1 | 1 | 1 | | |
| | | | | | 1 | | |
| | 1 | 1 | 1 | 1 | 1 | 1 | |
| | | 1 | 1 | 1 | 1 | | 1 |

Intersection with its transpose yields

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |
| | 1 | | | | | 1 | |
| | | 1 | 1 | 1 | | | |
| | | 1 | 1 | 1 | | | |
| | | 1 | 1 | 1 | | | |
| | | | | | 1 | | |
| | 1 | | | | | 1 | |
| | | | | | | | 1 |

A= [1]

B= [2,7]

C= [3,4,5]

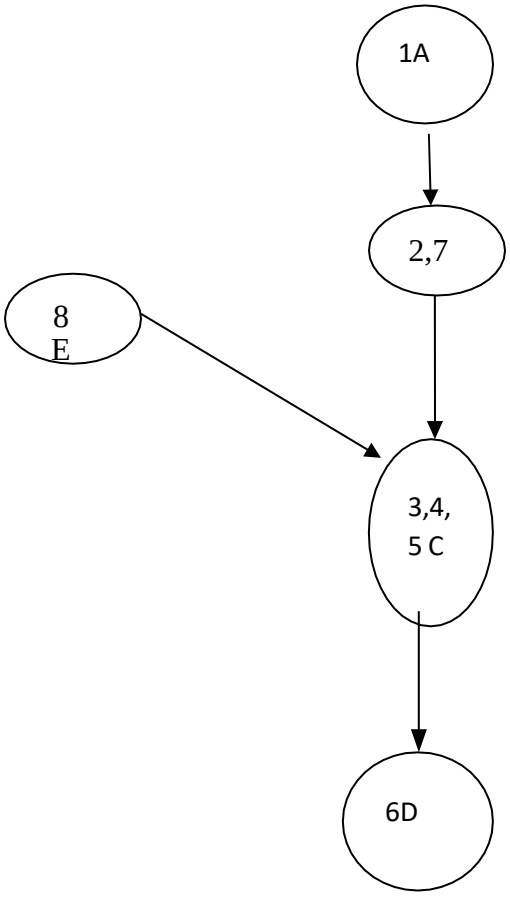
D= [6]

E=[8]

Whose graph is

A B C D E

| | | | | |
|---|---|---|---|---|
| 1 | 1 | | | |
| | 1 | 1 | | |
| | | 1 | 1 | |
| | | | 1 | |
| | | 1 | | 1 |



**Node-Reduction Algorithm**

Steps for node reduction algorithm

1. select a node for removal; replace the node by equivalent links that bypass that node and add those nodes links to the links they parallel.
2. Combine the parallel terms and simplify
3. Observe loop terms and adjust the outlinks of every node that had a self-loop to account for the effect of the loop.
4. the result is a matrix whose size has been reduced by 1. Continue until only the two nodes of interest exist.

The GET/RETURN problem

| | | | | | | |
|---|---|---|---|---|---|---|
| . | | G | | | | |
| | . | | | | | |
| | | . | G+R | | | |
| | | | . | G | | |
| | | | | . | G | |
| | | | | R | . | G |
| | R | | | | | .R |

| | | | | | | |
|---|---|---|---|---|---|---|
| . | | G | | | | |
| | . | | | | | |
| | | . | G+R | | | |
| | | | . | G | | |
| | | | | . | G | |
| | | | | R | . | G |
| | $R^+$ | | | | | . |

Because $R*R=R^+$

| | | | | | |
|---|---|---|---|---|---|
| . | | G | | | |
| | . | | | | |
| | | . | G+R | | |
| | | | . | G | |
| | | | | . | G |
| | $GR^+$ | | | R | . |

| | | | | |
|---|---|---|---|---|
| . | | G | | |
| | . | | | |
| | | . | G+R | |
| | | | . | G |
| | $G^2R^+$ | | | .1 |

| | | | |
|---|---|---|---|
| . | | G | |
| | . | | |
| | | . | G+R |
| | $G^3R^+$ | | . |

| | | |
|---|---|---|
| . | | G |
| | . | |
| | $(G+R)G^3R^+$ | . |

| | |
|---|---|
| . | $G(G+R)G^3R^+$ |
| | . |

$(G^2+GR)\ G^3R^+=(\ G^2+1)\ G^3R^+$
$=(G^4+\ G^2)R^*$

- Students understand the necessity of automation testing , what are the automated tools and how they were working, and classification of automated testing tools.

- ***Topic: Test Automation***

- ***Motivation (Why this topic is significant for the discussion?):***
- For repetitive process and complex application it is difficult to do manualtesting so move towards to automation testing.

- ***Notes:*** **Introduction**
  **Definition of automated testing-** automating human activities in orderto validate the application is called Automation testing
  Automated testing performed using scripting languages or any one of thethird party automation tool like QTP, selenium, Win runner, Silk etc

Advantages

a. Fast in execution
b. More reliable

c. More consistency
d. Automation scripting is re-usable for different versions of builds tovalidate
e. Automation script is repeatable

Disadvantages

a. Automation tools are expensive
b. Skilled automation test engineers are required
c. Tools are not support for different environment application

Automated testing should not be viewed as replacement for manual testing. But everything not possible to automate there are many activities are therein the testing life cycle which cannot possible to automate

**Load runner**

**Performance Testing Goals:**

It is conducted to accomplish the following goals:

- Verify Application's readiness to go live.
- Verify if the desired performance criteria are met
- Compare performance characteristics/configurations of the applicationto what is standard
- Identify Performance bottlenecks.
- Facilitate Performance Tuning.

**Key Activities in Performance Testing:**

**#1. Requirement Analysis/Gathering**

Performance team interacts with the client for identification and gathering of requirement – technical and business. This includes getting information on application's architecture, technologies and database used, intended users, functionality, application usage, test requirement, hardware & softwarerequirements etc.

**#2. POC/Tool selection**

Once the key functionality are identified, POC (proof of concept – which is a sort of demonstration of the real time activity but in a limited sense) is done with the available tools. The list of available**performance test tools** dependson cost of tool, protocol that application is using, the technologies used to buildthe application, the number of users we are simulating for the test, etc. During POC, scripts are created for the identified key functionality and executed with 10-15 virtual users.

**#3. Performance Test Plan & Design**

Depending on the information collected in the preceding stages, test planning and designing is conducted.

Test Planning involves information on how the performance test is going to take place – test environment the application, workload, hardware, etc.Test designing is mainly about the type of test to be conducted, metrics to be measured, Metadata, scripts, number of users and the execution plan.

During this activity, a Performance Test Plan is created. This serves as an agreement before moving ahead and also as a road map for the entire activity. Once created this document is shared to the client to establish transparency on the type of the application, test objectives, prerequisites, deliverable, entry and exit criteria, acceptance criteria etc.

**Briefly, a performance test plan includes:**

a) Introduction(Objective and Scope)
b) Application Overview
c) Performance(Objectives & Goals)
d) Test Approach (User Distribution, Test data requirements, Workload criteria, Entry& Exitcriteria,Deliverable,etc.)
e) In-Scope and Out-of-Scope
f) Test Environment (Configuration, Tool, Hardware, Server Monitoring,Database,test configuration, etc.)
g) Reporting & Communication
h) Test Metrics
i) Role & Responsibilities
j) Risk & Mitigation
k) Configuration Management

**#4. Performance Test Development**

- Use cases are created for the functionality identified in the test plan as the scope of PT.
- These use cases are shared with the client for their approval. This is to make sure the script will be recorded with correct steps.
- Once approved, script development starts with a recording of the steps inuse cases with the performance test tool selected during the POC (Proofof Concepts) and enhanced by performing Correlation (for handling dynamic value), Parameterization (value substitution) and custom functions as per the situation or need. More on these techniques in our video tutorials.
- The Scripts are then validated against different users.
- Parallel to script creation, performance team also keeps working onsetting up of the test environment (Software and hardware).
- Performance team will also take care of Metadata (back-end) through scripts if this activity is not taken up by the client.

**#5. Performance Test Modeling**

Performance Load Model is created for the test execution. The main aim of this step is to validate whether the given Performance metrics (provided by clients) are achieved during the test or not. There are different approaches to create a Load model. "Little's Law" is used in most cases.

**#6. Test Execution**

The scenario is designed according to the Load Model in Controller or Performance Center but the initial tests are not executed with maximum users that are in the Load model.

Test execution is done incrementally. For example: If the maximum number of users are 100, the scenarios is first run with 10, 25, 50 users and so on, eventually moving on to 100 users.

#7. Test Results Analysis

Test results are the most important deliverable for the performance tester. This is where we can prove the ROI (Return on Investment) and productivity that a performance testing effort can provide.

**Quick Test Professional**

QTP V 10.0

1. it is a product of HP (earlier it is Mercury Interactive)
2. it is a Functionality and Regression testing tool
3. it is compatible with window OS only
4. it supports client/server and web based applications to automate like Java, .Net, PHP, HTML, SAP, Multimedia, Mainframes (Terminal Emulator), XML, Oracle applications, Delphi, people soft, Visual Basic, Active X controls , VC ++, ASP.Net…etc
5. QTP supports *VB Script and java scripts for automation script

6. basic working principle is “Record & Play Back”
    i. Record: by default QTP able to convert our business transactions which we perform on AUT into automation script (in Vb Script)
    ii. Play back: during script runtime QTP will perform same operation on AUT w. r. to script
7. available versions in QTP 6.0, 7.0, 8.0, 8.2, 9.2, 9.5 , 10.0 (jan 2009), 11.0

Components in QTP Main screen:

1. Tool Bar:

It contains menu options and icons to perform operations on QTP

2. Test Pane:

It is like an Editor screen, where we can generate automation script and wecan perform some editing operations

Automation script we can generate using recording modes in QTP or bywriting script manually

In Test Pane there are 2 types of views:

a. Expert View:
In this view by default script generates in VB Script

b. Keyword View:
In this view script generates in simple understandable language in termsof “Item”, “Operation”, “Value” and “Documentation”

Note: in general we prefer “Expert View” where it is easy to write the script manually

Keyword view easy to understand the script without Vb script knowledge

3. Active Screen:

During recording by default QTP captures snapshot of application for eachoperation and those will be maintained in Active screen component

Advantages of active screen:

a. easy to understand script

b. we can perform some editing operations in the script like insertingcheckpoints, output values, new steps…etc
c. we can view/add test object properties into Object Repository

**Disadvantage:**

Active screen files will occupy more memory space

4. Data table:

in QTP we have built-in data table where we can import/store required testdata and from that we can parameterize test script during runtime

*there are 2 types of sheets in Data table

a. Global sheet

b. Action/Local sheet

Note: in QTP within the Test we can create maximum 255 Actions, for eachaction QTP provides individual action sheets in Data table

a. Global sheet:

By default script will execute multiple times based on number of rows filled with test data in Global sheet using test data from Global sheet we can parameterize any action script

b. Action/local Sheet:

Irrespective of number of rows filled with test data in Action/local sheet script will execute only one time

Using test data from Action sheet we can parameterize that particular action script only

5. Test Flow:

In this component we can view the sequence of actions execution flow in a test and also we can re-order those actions execution flow by performing Drag and Drop option

In general Actions will execute, in which sequence we created those actions in a test

6. Debug Viewer:

During execution break time to view the intermediate values of variablesand to update those values we use Debug viewer

In Debug viewer we have 3 sections

a. Watch  to view specific variable value
b. Variables  to view all the variable values
c. Command to update the value in a variable

7. Information pane: (Ctrl+F7)

This component will provide syntax error information in the script

8. Missing Resources:

In general for a test we associate different resource files like shared repositories, recovery scenarios, environment variables, test data, library functions…etc

For a opened test if any associated resource file is not available that information we can find in "Missing Resources" component

- Generating Basic Script:

Based on automation scenario we generate basic script to perform operations on AUT. We can generate basic script using recording modes in QTP or by writing script manually

1. verify multiply functionality in "Calculator" applicationProcedure:

Click on "C"

Click on "5"

Click on "*"

Click on "6"

Click on "="

Expected Result= 30

Navigation:

Open New Test in QTP Click

on "Record" icon

Select option in "Record & Run settings" based on requirementClick on

"OK"

Perform required operations on AUT which you want toconvert into automation script

Stop recording

Checkpoints:

Using checkpoints we can verify actual values in AUT w. r. to our expectedvalues

There are 10 checkpoints in QTP

1. Standard checkpoint

2. Text checkpoint
3. Text Area Checkpoint
4. Bitmap checkpoint
5. DB checkpoint
6. XML checkpoint
7. Accessibility checkpoint
8. Image checkpoint
9. Table checkpoint
10. Page checkpoint

Basic working principle of checkpoint:

Step 1: while creation time of checkpoint TE should provide his expectedvalue to the checkpoint

Step 2: during Runtime checkpoint captures actual value from AUT

Step 3: checkpoint compares expected and Actual values based on that itprovide status into test result

Where as

Expected Result = = Actual Result □PassedExpected

Result! = Actual result □Failed

**Selenium**

**Brief Introduction Selenium IDE**

Selenium Integrated Development Environment (IDE) is the simplest frameworkin the Selenium suite and is the easiest one to learn. It is a Firefox plug-in that you can install as easily as you can with other plug-in. However, because of its simplicity, Selenium IDE should only be used as a prototyping tool. If you want to create more advanced test cases, you will need to use either Selenium RC or WebDriver.

**Selenium Grid**

Selenium Grid is a tool used together with Selenium RC to run parallel tests across different machines and different browsers all at the same time. Parallel execution means running multiple tests at once.

**Features:**

- □ Enables simultaneous running of tests in multiple browsers and environments.
- □ Saves time enormously.
- □ Utilizes the hub-and-nodes concept. The hub acts as a central source ofSelenium commands to each node connected to it.

**Note on Browser and Environment Support**

Because of their architectural differences, Selenium IDE, Selenium RC, and WebDriver support different sets of browsers and operating environments.

**How to Choose the Right Selenium Tool for Your Need**

| Tool | Why Choose? |
|---|---|
| **Selenium IDE** | • To learn about concepts on automated testing and Selenium, including:<br>  ◦ Selenium commands such as type, open, click And Wait, assert, verify,etc.<br>  ◦ Locators such as id, name, xpath, css selector, etc.<br>  ◦ Executing customized JavaScript code using run Script<br>  ◦ Exporting test cases in various formats.<br>• To create tests with little or no prior knowledge in programming.<br>• To create simple test cases and test suites that youcan export later to RC or WebDriver.<br>• To test a web application against Firefox only. |
| **Selenium RC** | • To design a test using a more expressive languagethan Selenium<br>• To run your test against different browsers (exceptHtmlUnit) on different operating systems.<br>• To deploy your tests across multiple environmentsusing Selenium Grid.<br>• To test your application against a new browser thatsupports JavaScript.<br>• To test web applications with complex AJAX-basedscenarios. |
| **WebDriver** | • To use a certain programming language in designingyour test case.<br>• To test applications those are rich in AJAX-based functionalities.<br>• To execute tests on the HtmlUnit browser.<br>• To create customized test results. |
| **Selenium Grid** | • To run your Selenium RC scripts in multiple browsers and operating systems simultaneously.<br>• To run a huge test suite, that need to complete insoonest time possible. |